Assessing and Improving the Mutation Testing Practice of PIT

Thomas Laurent*†, Anthony Ventresque*, Mike Papadakis‡, Christopher Henard‡, and Yves Le Traon‡
*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 used extensively to support the experimentation of software engineering studies. Its application to real-world projects is possible thanks to modern tools that automate the whole mutation analysis process. However, popular mutation testing tools use a restrictive set of mutants which do not conform to the community standards as supported by the mutation testing literature. This can be problematic since the effectiveness of mutation depends on its mutants. We therefore examine how effective are the mutants of a popular mutation testing tool, named PIT, compared to comprehensive ones, as drawn from the literature and personal experience. We show that comprehensive mutants are harder to kill and encode faults not captured by the mutants of PIT for a range of 11% to 62% of the Java classes of the considered projects.**

I. Introduction

Software testing constitutes the current practice for checking programs. In such a scenario, sets of test cases are selected and used to examine the behavior of the programs under investigation. To quantify the "quality" of the test cases, researchers and practitioners use the so-called adequacy metrics or testing criteria [1]. These metrics measure the quality achieved by the employed test sets.

Mutation analysis is an established test criterion [1], [2] that promises to thoroughly examine the programs under investigation. It operates by evaluating the ability of the candidate test cases to distinguish between the program under test and a set of altered program versions, called mutants. Mutants represent program defects and are used to measure the ability of the test cases to reveal them. The power of the technique is based on the ability of the mutants to represent real faults [3], [4] and to lead testers in writing test cases that cover almost all the other test criteria [5], [6], [7].

The downfall of mutation is its application cost. This is related to the number of possible mutants which can be prohibitively high [8], [9]. Each mutant forms a different program version that needs to be executed with the candidate test cases. Therefore, a large number of test executions is required in order to compute the adequacy measurement.

Because of the large number of mutants, practitioners believed that mutation does not scale to real-world programs. However, modern tools proved this belief wrong and as a result mutation "entered the mainstream" of practice [5]. To this end, several tools have been developed, linked with build systems and development tools. Modern tools are also robust and they can easily be used by developers [10]. As a result, they are used extensively in software engineering studies.

Unfortunately, popular tools like PIT [11] 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 [12]. Since these tools are extensively used in software engineering studies, it is mandatory to validate the extent to which their adopted mutant set is representative of the community standards as supported by the mutation testing literature.

This paper presents a thorough study investigating the above-mentioned issue using PIT [11]. We use PIT since it was found to be the most robust available mutation testing tool [10] and it has been used extensively for research purposes in the recent years, e.g., [13], [14], [15], [16]. Our study indicates significant limitations of the popular mutants and thus, motivate the need for a more comprehensive one.
In summary, the contributions of this paper are:

- We describe and implement a comprehensive mutant set for Java that reflects the beliefs of the mutation testing community as it has been recorded in the literature [9], [4] and discussed during the Mutation 2014 and 2015 workshops, e.g., [17].
- We provide empirical evidence that the comprehensive mutant set is superior to the one often used by mutation testing tools. This set is statistically significant superior from 11% to 62% of the studied program classes.
- To support future research, we will submit our code to the PIT repository to became available. Our new version of PIT that supports the comprehensive mutants is also available or request[1].

II. Terminology & Background

This section introduces the terminology and the concepts that are used throughout the paper. First, II-A presents the mutation testing process. Then, Section II-B describes the selection of mutants. Finally, the notion of disjoint mutants is introduced in Section II-C.

[1]For inquires please contact Anthony Ventresque, anthony.ventresque@ucd.ie

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 variant 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 language operator such as the '$+$' to another one, such as '$-$'.

Mutants are produced by syntactic changes introduced by mutant operators. These are the instances that are produced by applying an operator on every point of the code under investigation that matches their respective rule. Thus, every mutant has a single and specific syntactic difference from the original program. For reasons that we will discuss in the related work (see Section VI), mutation testing uses mutants produced by simple syntactic changes.

Mutants are used to measure how good the employed test cases are in checking the software under assessment. This is done by observing the runtime behavior of the original, non-mutated, and the mutated programs. When comparing the program outputs of the original with the mutated programs, and found differences, we exhibit behavior discrepancies [18]. Such differences are attributed to the ability of the used test to project the syntactic program changes to its behavior, i.e., to show a semantic difference. When mutants exhibit such differences in their behavior, they are called "killed". Those that do not exhibit such differences are called "live". Mutants might not exhibit any difference in their behavior either because the employed test cases were not capable of revealing them or because they are functionally equivalent with the original program. Mutants belonging to the latter case are called equivalent [18].

Mutation testing refers to the process of using mutation analysis as a means of quantifying the level of thoroughness of the test process. Thus, it 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. Ideally, to have an accurate metric, equivalent mutants must be removed from the calculation of the score. However, this is not possible since judging programs' equivalence is an undecidable problem [19].

B. Mutant Selection

Selective mutation was shown to be valid in several studies involving programs written in Fortran [9] and in C [20]. As a result, Java mutation tools were built based on the findings of these studies. To address the scalability issues of the method, tool developers made further reductions. Thus, popular tools like PIT [11] support a very small and restrictive set of mutants that neither follows any suggestion from previous studies nor practical experience.

PIT, even in its latest version that supports an extended mutant set, has several shortcomings. One such example is the relational operator for which PIT replaces one instance of the operator by only another one, i.e., mutates $<$ only to $<=$, or $<=$ only to $<$, or $>$ only to $>=$, or $>=$ only to $>$. However, this practice is not sufficient. Indeed, previous studies have shown that three mutants are needed to avoid a reduced effectiveness of the method [21], [22].

Although practical, mutant reduction should not be at the expense of the method effectiveness. Almost all previous studies were based on the assumption that mutants are equal [23]. However, this does not hold in practice and has the potential to bias the conducted research as recent studies show [24], [23], [25]. 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 [3], [8], i.e., it is used to measure the level of test thoroughness achieved by various testing methods. Mutation score serves as a comparison basis between testing techniques and hence, as a yardstick to judge the winning one. This practice is quite popular, and introduce severe problems that can threaten the validity of the conducted research.

The problem is that not all mutants are of equal power [23], which means that some are useful and some are not. Indeed, mutants cover the full spectrum of cases, including trivial ones, very easy to kill, duplicated, equivalent ones and also hard to kill ones. Those of the last category are of particular interest since they lead to strong tests [4], [23]. Hard to kill, trivial and easy to kill mutants are defined with respect to the employed test suite [23]. 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 [26]. However, this process might not be adequate since in most cases many mutants tend to be killed jointly [25]. 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[2].

This issue was initially raised by Kintis *et al.* [25] who introduced the concept of *disjoint mutants*, i.e., minimum number of mutants that contribute to mutation score. Their use is motivated by the same study which demonstrated that hard to kill mutants also suffer from the inflation problem. Later Amman *et al.* [24] formalized this concept, name it as "minimum mutants", and suggested using it as a way to bypass the mutation score inflation problem.

In this paper we follow an analysis based on both all and disjoint mutants. Disjoint mutants have the advantage to cover the whole spectrum of mutants and suffer less from the mutant inflation effect. Their identification is an NP-complete

[2]Kintis *et al.* [25] reports that this is 9% of mutants, for Java programs using the muJava mutation testing tool, Amman *et al.* [24] report 10% for the Java mutants of muJava tool and 1% for the C mutants of the Proteum tool.

Algorithm 1: Disjoint Mutants

```
Input: A set $S$ of mutants
Input: A set $T$ of test cases
Input: A matrix $M$ of size $|T| \times |S|$ such as $M_{ij} = 1$ if test$_i$ kills mutant$_j$
Output: The disjoint mutant set $D$ from S
1  $D = \emptyset$
                                            /* Remove live mutants */
2  $S = S \setminus \{m \in S \mid \forall i \in 1..|T|, M_{ij} \neq 1\}$
                                       /* Remove duplicate mutants */
3  $S = S \setminus \{m \in S \mid \exists m' \in S \mid \forall i \in 1..|T|, M_{ij(m)} = M_{ij(m')}\}$
4  while ($|S| > 0$) do
5      maxSubsumed = 0
6      subsumedMut = null
7      maxMutSubsuming = null
                              /* Select the most subsuming mutant */
8      foreach ($m \in S$) do
9          sub$_m$ = $\{m' \in S | \forall i \in 1..|T|, (M_{ij(m)} = 1) \Rightarrow (M_{ij(m')} = 1)\}$
10         if ($|$sub$_m| >$ maxSubsumed) then
11             maxSubsumed = $|$sub$_m|$
12             maxMutSubsuming = $m$
13             subsumedMut = sub$_m$
14         end
15     end
                            /* Add the most subsuming mutant to $D$ */
16     $D = D \cup \{$maxMutSubsuming$\}$
           /* Remove the subsumed mutants from the remaining */
17     $S = S \setminus$ subsumedMut
18 end
19 return $D$
```

problem [24] and thus, we use a greedy approximation method. Algorithm 1 details their computation from a set of mutant S. First, the live and duplicate mutants are removed from S (lines 2 and 3). Then, the most subsuming mutant is retrieved (lines 8 to 15). It is the mutant which, when killed, implies the highest number of other mutants to be killed as well. This mutant is then added to the disjoint set D (line 16) and the subsumed mutants are removed from S (line 17). This process is repeated until S is empty. Finally, the set of disjoint mutants, D, is returned.

III. Motivation

Mutation testing is extensively used by researchers and has an increasing use by practitioners and the open source community [8], mainly due to the existence of automated tools. However, mutation is sensitive to the set of mutants that are used [12]. Therefore, it is mandatory to equip these tools with a comprehensive set of mutants that can adequately measure test thoroughness.

In this paper, we deal with this issue by investigating the extent to which the mutant sets employed by popular mutation testing tools meet the standards as expressed by the mutation testing literature and community. We call the first set as the “common” mutant set and the second one as the “comprehensive” one.

Our goal is to validate the use of the popular mutant set which was introduced by the developers of popular mutation testing tools. We seek to investigate this issue since modern tools like PIT [11] have been extensively used in the recent years[3]. Thus, a possible issue with their adopted mutants can question the effectiveness of the mutation method and hence the conducted research. We therefore compare the extent to which the popular mutant set conforms to the test requirements possessed by the mutation testing literature. To validate this practice we use large open source projects written in Java with mature test suites.

[3]For instance, [13], [14], [15], [16] are recent publications that use PIT.

IV. Experimental Study

This section first states the Research Questions (RQs) under investigation. Then, the subjects, settings and tools used for the experiments are described. Finally, the last subsections detail the studied mutant operators and the analysis procedure followed to answer the RQs.

A. *Definition of the Experiment and Research Questions*

Current research on software engineering has largely focused on using mutation analysis 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 employed mutants [12]. Therefore, it is important to know whether the commonly used mutants, as supported by these tools, are suitable. In other words we seek to determine the degree to which the commonly used mutants are representative of those suggested by the literature, i.e., the comprehensive mutant set. This leads us to our first research question:

RQ1 **(Effectiveness)**. Is there any effectiveness difference between the commonly used mutants and the comprehensive ones?

Since we are interested in testing, we seek to identify the mutants that are more effective at measuring the ability of test cases to exercise each point of the program under investigation. Mutation score measurements can differ when different mutant sets are employed. The measurements are affected by the number of: mutants, of equivalent ones, of trivial and hard to kill ones. To deal with this issue, we perform an objective comparison, i.e., we measure the extent to which one method covers the requirements of the other, between the two examined mutant sets. Thus, we seek to measure the ratio of mutants, of the one set, that are found by the tests that are selected based on the other mutant set. Thus, the ”weaker” mutants will lead to ”weaker” tests and hence kill a smaller fraction of the ”stronger” mutants. Here, it should be noted that objective comparisons form a common practice in mutation testing literature, e.g., [4], [6], [9].

As discussed previously, in Section II-C, there is a potential problem with this practice due to the inflation effect of the trivial mutants. We thus measure the ratios of the disjoint mutants that are killed. Therefore, to answer RQ1 we report results based on two effectiveness measures; the percentage of all mutants killed and the percentage of the disjoint ones that are killed.

So far, our investigations focus on whether mutants of one set can capture all the faults introduced by the other set. However, this analysis tells us nothing about the difficulty of exposing mutants. Thus, mutant easiness is another important attribute of mutants [4], [23]. This is due to the fact that hard to kill mutants indicate a relatively small semantic

TABLE I
SUBJECTS USED IN THE EXPERIMENTS. THE REPORTED 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	210	4,129
jfreechart	1.0.19	46,986	290	1,320
jaxen	1.1.6	6,790	152	646
commons-lang	3.3.4	16,286	199	3,373
commons-collections	4.4.0	11,281	243	2,210

change difference [23], [27] that is often easy to overlook when testing. Thus, through our second research question, we investigate whether the comprehensive mutant set introduces harder to kill mutants than the commonly used one:

RQ2 **(Easiness)**. What is the difference, in terms of difficulty to expose mutants, between the common and the comprehensive mutants?

Mutation testing has a widespread reputation of being computationally demanding. In the past, practitioners believed that it cannot scale to real-world systems mainly due to the large number of mutants. However, tools like PIT and Javalanche proved that this belief was incorect [28]. This ability of the tools can be attributed to the restricted mutant set they employ and to the advanced mutant generation and execution techniques used. Therefore, it is possible that the comprehensive mutant set is too expensive to be used in practice. Hence, we investigate:

RQ3 **(Scalability)**. What is the execution time differences of the comprehensive mutants when compared with the common ones?

We measure execution time since it forms a direct measure of the application cost of the method. We do not consider other parameters that can influence the application cost, such as the number of equivalent mutants, the test generation cost, since they fall outside the scope of the present paper. Here, we focus on the effectiveness of the method as conducted by recent studies and thus, leaving the issue of its application cost open for future research.

B. Subject Programs

The experiments are conducted on the 5 Java projects recorded in Table I. For each of them, the version, lines of code (calculated with the JavaNCSS tool [29]), number of classes (for which test suites exist) and number of tests are reported.

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. Commons-lang provides a set of utility methods for the commons classes of Java. Finally, commons-collections provides data structures in addition to those existing in the standard Java framework.

C. Experimental Environment

All the experiments were performed on a quad-core Intel Xeon processor (3.1GHz) with 8GB of RAM and running Ubuntu 14.04.3 LTS (Trusty Tahr).

D. Employed Tools

We use PIT, a popular mutation testing tool, to support our experiments. We use the 1.1.5 release with the extended set of mutants that it supports. To enable a comparison with the comprehensive mutants, we modified PIT to support them. The next section details all the considered mutants.

E. Employed Mutants

The employed mutant sets are described in Table II. The common mutants are described in the upper part of the table while the comprehensive ones are described in the lower part. The comprehensive mutant set was formed based on the beliefs of the mutation testing community [17] and the literature. In particular we adapt to Java the set of mutants that was suggested and used in the following studies [9], [4], [2].

Note that the comprehensive set of mutants includes all the common ones. For each mutant, Table II records its name, a description of the transformation performed and an example.

Special care was taken in order to reduce the duplicated mutant instances [26] by removing the overlap between the operators. It is also possible that some mutants might be redundant [30]. However, using an analysis similar to [30] may degrade the effectiveness of the method in cases of mutants that cannot be propagated. We discuss this issue in the related work section, i.e., VI-C. To avoid such risk we rely on disjoint mutants to remove redundancies among the mutants.

Table III presents some descriptive statistics (minimum median, mean and maximum values) about the employed mutants as they appear in the classes of the studied projects. Thus, the table records details about the number of mutants, the number of killable mutants (determined based on the available test suite), and mutation score for the common and comprehensive operators for each project.

F. Analysis Procedure for Answering the Research Questions

To answer RQ1, we constructed test suites using the common set. This was performed by incrementally adding random tests in the suites and keeping only those that increase mutation score. So, if the randomly selected tests failed to kill any additional mutant, i.e., it is redundant with respect to the employed mutants, the test was not included. This is a typical process followed by many previews studies, e.g., [4], [7]. Thus, we measured (a) the number of mutants of the comprehensive set that are killed by the tests selected based on the common set and (b) the number of mutants of the comprehensive sets found when using all available tests. Since the tests were selected at random, this process was repeated 30 times. As a result, we obtain 30 instances for each one of the two measures for every class of each project. We compared them with Wilcoxon test using the R statistical computing project [31]. From this test, we obtain a p-value which represents the probability that measure (a) is higher than measure (b). Following the usual statistical inference procedures we consider the differences as statistically significant if they provide a p-value lower than

TABLE II
COMMON AND COMPREHENSIVE MUTANTS.

	Name	Transformation	Example	Name	Transformation	Example
Common op.	**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` $\rightsquigarrow$ `return 1`
	Negate Cond.	Negates one relational operator (single negation).	$== \rightsquigarrow !=$	**Void Meth. Call**	Deletes a call to a void method.	`void m()` $\rightsquigarrow$
	Remove Cond.	Replaces a cond. branch with true or false.	`if (...)` $\rightsquigarrow$ `if (true)`	**Meth. Call**	Deletes a call to a non-void method.	`int m()` $\rightsquigarrow$
	Math	Replaces a numerical op. by another one (single replacement).	$+ \rightsquigarrow -$	**Constructor Call**	Replaces a call to a constructor by null.	`new C()` $\rightsquigarrow$ `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` $\rightsquigarrow$ `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.	$1 \rightsquigarrow 0, a \rightsquigarrow a+1$			
Comprehensive op.	**ABS**	Replaces a variable by its negation.	$a \rightsquigarrow -a$	**OBBN**	Replaces the operators & by $\vert$ and vice versa.	$a\&b \rightsquigarrow a \vert b$
	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 AND COMPREHENSIVE MUTANTS.

		joda-time		jfreechart		jaxen		commons-lang		commons-collections	
Measure		Common op.	Compre. op.	Common op.	Compre. op.	Common op.	Compre. op.	Common op.	Compre. op.	Common op.	Compre. op.
#Mutants	*Min.*	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Med.	97.00	224.00	98.00	260.50	24.00	39.00	27.00	57.00	27.00	42.00
	Mean	164.17	462.06	219.14	685.48	77.48	188.97	156.82	457.05	62.32	126.53
	Max.	973.00	2,915.00	3,436.00	9,742.00	3,901.00	14,493.00	4,545.00	14,586.00	1,094.00	2,349.00
#Killable	*Min.*	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Med.	60.99	136.99	26.00	49.00	11.99	21.00	17.00	33.50	5.00	5.00
	Mean	117.32	295.71	59.59	131.20	37.91	69.31	124.74	338.86	21.66	41.34
	Max.	834.00	2,108.00	1,356.00	2,488.00	773.00	1,793.00	3,928.99	11,522.99	867.00	1,553.00
MS	*Min.*	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Med.	0.80	0.71	0.24	0.16	0.73	0.66	0.84	0.74	0.50	0.45
	Mean	0.71	0.64	0.29	0.24	0.60	0.56	0.72	0.66	0.44	0.41
	Max.	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

0.05, i.e., 5% is our significance level. In RQ1, we record the number of classes for which there is a statistically significance difference. The ratio (a)/(b) forms the objective comparison score when using all mutants. The values of Table III imply that the projects have classes with only 1 mutant and 0 killable ones. Also, there are classes where all mutants are killed. Thus, we base the results of the objective comparison at the class granularity level and present them according the first three quartiles, i.e., according to the ordered 25%, median (50%) and 75% values. Finally, we compute the ratio of the disjoint mutants of the comprehensive set that are found by the tests selected based on the common set. The distance from value 1 on both the objective comparison and disjoint mutants scores quantify the effectiveness differences between the examined mutants.

To answer to RQ2, we measure the easiness of killing mutants. The easiness of killing a mutant is defined as the number of test cases that kill a mutant, towards the total number of test cases. As a result, when 100% of the test cases kill a mutant, the latter is denoted as very easy to kill.

To answer RQ3, we applied mutation analysis as it is supported by the current version of the tool and record the time required.

V. RESULTS & ANSWERS TO THE RESEARCH QUESTIONS

This section reports on the experimental results and answers the RQs stated in previous section.

A. RQ1 - Effectiveness

For this question, we first consider the number of comprehensive mutants killed by test cases selected based on the common mutants, and the number of comprehensive mutants killed by test cases targeting them. This forms two mutation scores for each Java class of the considered projects. The comparison has been performed 30 times, thus yielding 30 mutation scores of the two types per class.

Table IV records the classes for which there is a statistical significance between the two measures. As can be seen, there

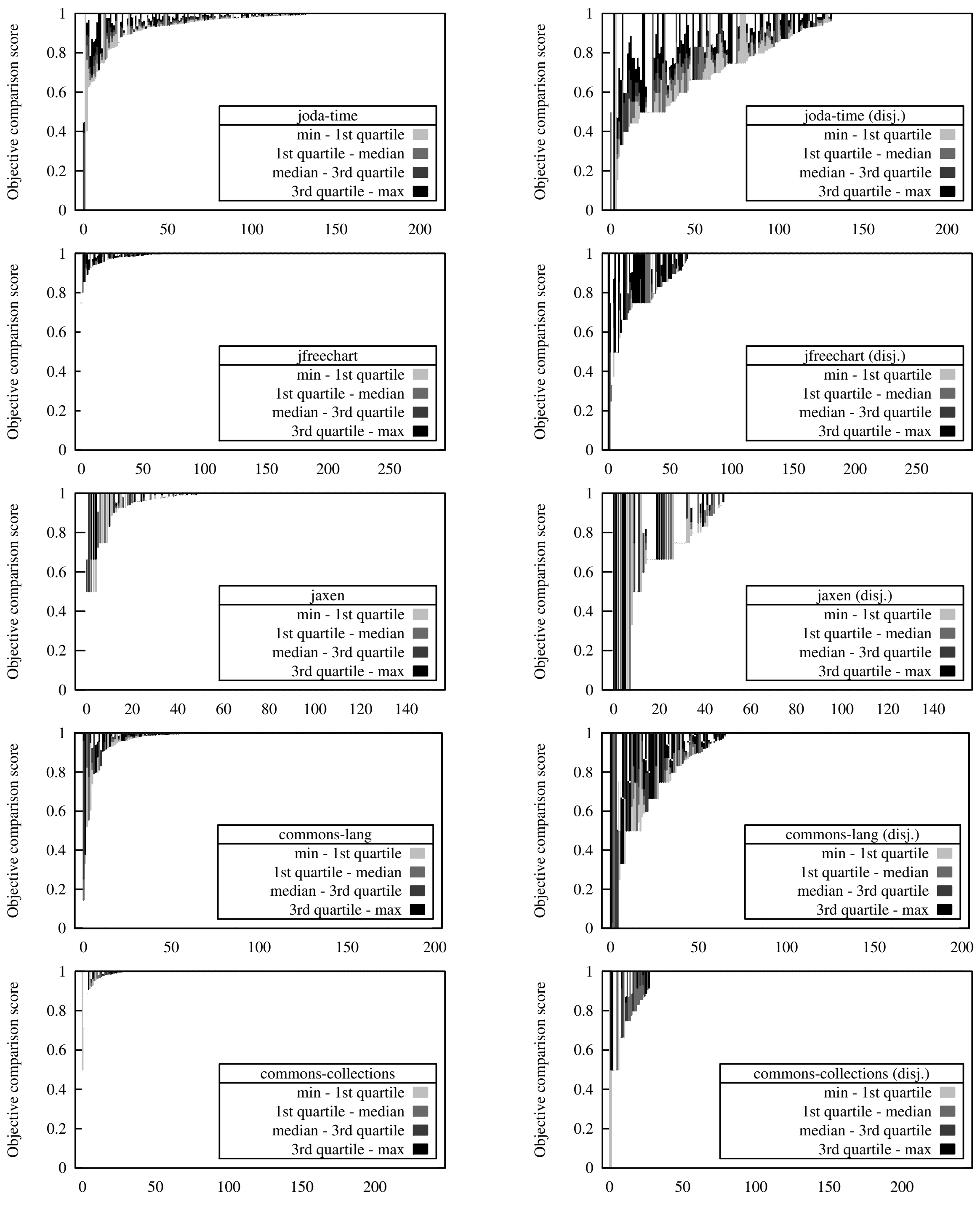

Fig. 1. Objective comparison results (RQ1 - effectiveness). The plots on the left side display the results of all mutants while on the right side the results of the disjoint mutants. The y-axis represents the distribution of the 30 scores per class, i.e., the minimum, first quartile, median, third quartile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

is a significant difference for 11% of the classes for commons-collections, for 62% of them for joda-time.

We now evaluate the percentage of mutants of the comprehensive set that are killed by the test cases selected based on the common set. This percentage is denoted as objective comparison score. Figure 1 records the objective comparisons scores for the 5 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 colors of the plot represent the distribution of the 30 ordered scores according to the quartiles. Thus, from the lightest to the darkest color, 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

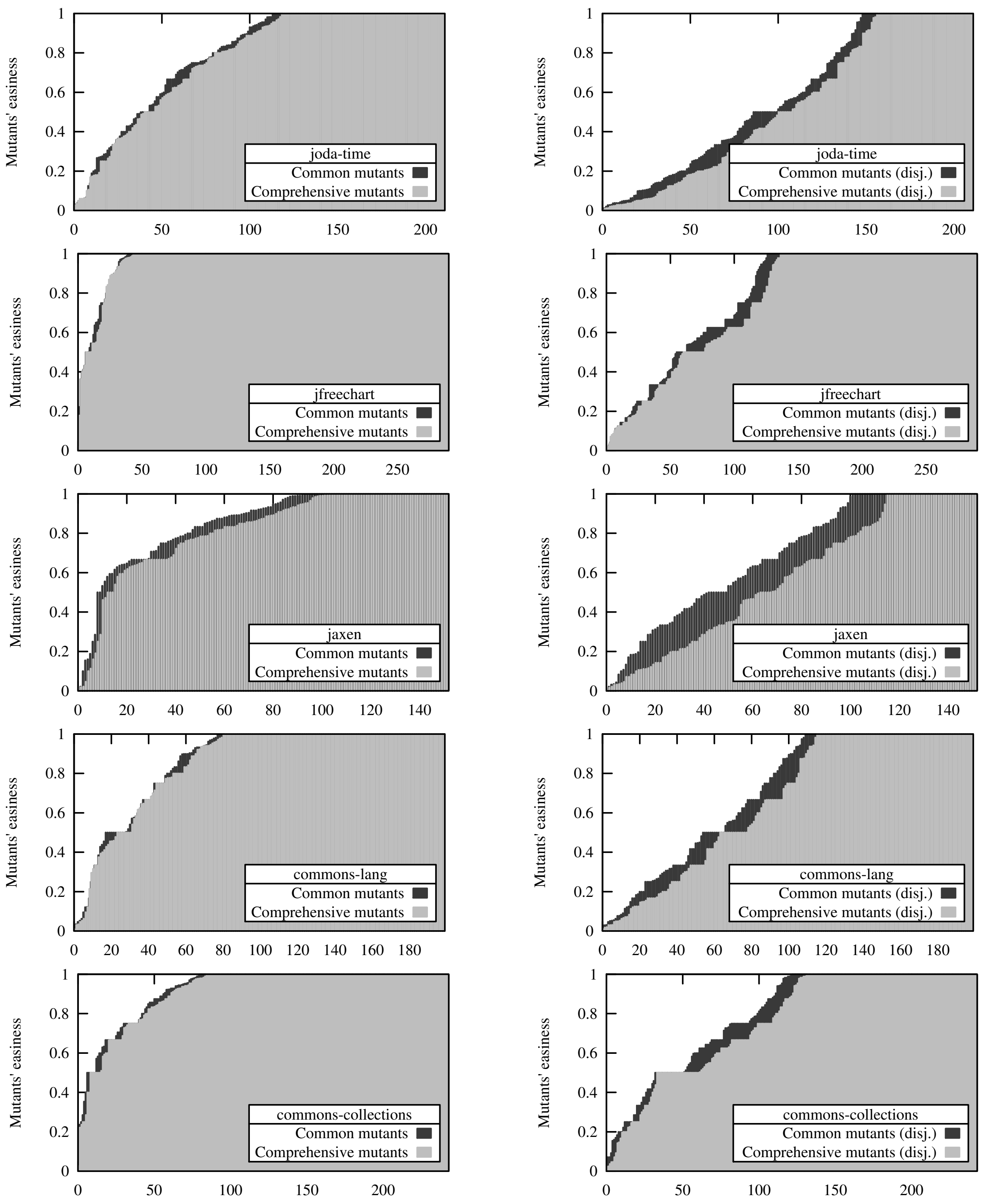

Fig. 2. Easiness of killing mutants (RQ2 - easiness). The plots on the left side display the easiness of all mutants while on the right side they display the easiness of the disjoint mutants. The y-axis represents the median mutant easiness, while x-axis represents the Java classes of the programs.

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 test cases are the same on both sets, while the presence of darker colors on the graph indicates that there are mutants that are missed by the test cases.

The plots on the left part of Figure 1 represent the scores when considering all the mutants while the plots on the right side depict the results when considering the disjoint mutants. On the left side of the figure, i.e., when all the mutants are considered, we can see that in most of the cases the scores are close to 1, which means that there are only a few mutants missed by the test cases. This is especially the case

TABLE IV
NUMBER AND PROPORTION OF CLASSES FOR WHICH THERE IS STATISTICAL SIGNIFICANCE BETWEEN THE COMMON AND COMPREHENSIVE MUTANT SETS.

Subject	#Classes (proportion)
joda-time	130 (62%)
jfreechart	64 (22%)
jaxen	43 (28%)
commons-lang	63 (32%)
commons-collections	26 (11%)

for jfreechart and commons-collections. For the other projects, there is a larger proportion of classes with lower scores, indicating that there are more mutants of the comprehensive sets missed by the test cases. Considering now the right part of Figure 1 that concerns the disjoint mutants, we can see that the proportion of missed mutants is even more important, in particular for joda-time where mutants are missed in more than half of the classes.

To conclude, there is a significant difference between the common mutants and the comprehensive ones. This shows that the test cases miss many killable mutants from the comprehensive set. The difference is even more significant when considering disjoint mutants.

B. *RQ2 - Easiness*

Here, we evaluate whether mutants are difficult to kill or not. The easiness of killing a mutant is the percentage of the test cases that kill this mutant. Thus, if all the test cases kill a given mutant, the easiness is 1. By contrast, an easiness close to 0 means that the mutant is very difficult to kill, since only few test cases are able to identify it.

Figure 2 shows the easiness, median values, of the common mutants against the comprehensive ones for each class. A greater surface indicates that the corresponding mutants are easier to kill. The plots on the left part consider all the mutants while the right side are for the disjoint mutants. From these results, we can observe that the comprehensive mutants (represented by the gray bars) are harder to kill. For all the mutants, the difference in terms of easiness compared to the common mutants range from 2-5% for the 50% of the commons-lang program classes. In the case of jaxen, the easiness difference is 12% in almost the 60% of the program classes. With respect to the disjoint mutants, the difference goes beyond 20% for approximately 75% of the jaxen classes.

To summarize, the comprehensive mutants are harder to kill than the common ones when considering either all the mutants or the disjoint ones only. It means that the comprehensive set introduces faults which are more difficult to expose.

C. *RQ3 - Scalability*

To evaluate the cost of using the comprehensive mutants, we measure the execution time of both the common and comprehensive sets. Table V records the number of mutants, killable mutants, execution time in seconds and execution time per mutant in seconds for both sets of mutants.

Considering the time in seconds, the comprehensive mutants require more time than the common ones. This is natural since the number of mutants is also much higher. The highest execution time is for jaxen with 31,077 seconds, i.e., approximately 8 hours and a half. Focusing on the time per mutants, the execution times between the two sets become rather similar, except for jaxen where the execution time is approximately 10 times longer.

Overall, the comprehensive set of mutants has an overhead in terms of execution time, but this is not unbearable.

VI. RELATED WORK

The following sections present studies with respect to mutation testing (VI-A), mutant reduction techniques (VI-B) and the accuracy of mutation score (VI-C).

A. *Mutation Analysis*

Mutation analysis was initially introduced as a test method helping the generation of test cases [32]. However, in recent years it has been proven to be quite powerful and capable of supporting various software engineering tasks [5]. In particular mutants have been used to guide test generation [22], [33], test oracle generation [33], to assist the debugging activities [18], [34], to evaluate fault detection ability [4] and to support regression testing activities like test selection and prioritization [16], [35]. The method has also been applied to models [36], software product lines [37] and combination strategies [38].

One of the main issues of the method is equivalent mutants [39], [40]. Despite the efforts from the community, e.g., [8], and recent advances [41], [26], [42], the problem remains open [26]. Similar situation arises when considering mutation-based test generation [22], [33].

Mutation has become popular [8] thanks to its ability to represent real faults [4]. Also, many modern mutation tools are integrated with build systems and development tools, thus making their application easy [10]. However, such tools usually support mutants that are more restrictive than the popular set, and hence, overestimate or underestimate the employed measures, as shown by the present paper. Previous research suggested that the comprehensive mutant set has the ability to prod-subsume and to reveal more faults than most of the other white-box testing criteria [5], [7].

B. *Mutant Reduction*

From the early days of mutation testing, it was evident that mutants were far too numerous to be used in practice. Therefore, researchers have tried to identify subsets of them that are representative. The first reduction was made towards the coupling effect hypothesis, which states that tests revealing simple mutants can also reveal complex ones [32], [43].

A straight way to reduce the number of mutants is to randomly sample them [44]. Although, sampling can provide a range of trade-offs, Papadakis and Malevris [44] provided evidence that mutant sampling ratios of 10% to 60% have a loss on fault detection from 26% to 6%, respectively. Similar results have been shown in the study of Wong *et al.* [45].

TABLE V
EXECUTION TIME IN SECONDS FOR THE COMMON AND COMPREHENSIVE MUTANTS (RQ3 - SCALABILITY).

	Common set				Comprehensive set			
Subjects	**Mutants**	**Killable**	**Time**	**Time/Mutant**	**Mutants**	**Killable**	**Time / Overhead**	**Time/Mutant**
joda-time 2.8.1	35,297	25,224	1,138	0.032	99,343	63,578	3,531 / 210%	0.035
jfreechart 1.0.19	81,960	22,289	2,398	0.029	256,370	49,069	6,589 / 175%	0.026
jaxen 1.1.6	14,334	7,014	1,221	0.085	34,960	12,823	31,077 / 2,445%	0.889
commons-lang 3 3.4	34,502	27,443	2,803	0.081	100,553	74,550	8,023 / 186%	0.080
commons-collections 4 4.0	24,308	8,449	570	0.023	49,354	16,126	1,230 / 116%	0.002

Other mutant reduction strategies fall in the category of selective mutation [9]. Selective mutation tries to reduce the arbitrariness of random sampling by using only specific types of mutants.

C. Duplicated, Trivial and Redundant Mutants

The presence of redundant mutants has long been recognized, i.e., since 1993 [46], but only recently, received attention. The studies of Tai [21], [46] were focused on reducing the application cost of fault-based testing strategies. This was based on constraints that restrict the mutants introduced by the relational and logical operators so that they only consider non-redundant ones. Thus, their suggestion was to restrict the mutant instances of the relational and logical operators to the minimum possible number. In a later study, Kaminski *et al.* [47], [48] came to a similar conclusion about the relational operator. Thus, they suggested that mutation testing tools should reduce the number of redundant mutants by restricting the mutant instances of the relational operator.

Papadakis and Malevris [49] suggested using minimized constraint mutant instances to efficiently generate mutation-based test cases. Thus, when aiming at generating test cases there is no point in aiming at non-redundant mutants. On the same lines, Just *et al.* [30], demonstrated that by constraining the relational and logical operators, it is possible to reduce some of the redundancy between the mutants.

All the approaches discussed so far were based on an analysis at the predicate level, specifically designed for "weak" mutation. Thus, when applied to strong mutation, the analysis may not hold. This can be due to the following two reasons; a) constructs like loops and recursion can make a statement to be exercised multiple times, and b) these approaches assume that when the identified mutants are killed the redundant ones are also killed. However, it is likely that the identified mutants are equivalent while the non-redundant ones are not, thus, resulting in degradation in the effectiveness of the method. These issues motivated the use of disjoint mutants that do not suffer from these problems.

The first study suggesting the use of non-redundant mutants when comparing testing techniques is that of Kintis *et al.* [25]. As discussed in Section II-C, Kintis *et al.* introduced the notion of disjoint mutants and demonstrated that the majority of mutants produced by the MuJava mutation testing tool is redundant. In the same lines, Amman *et al.* [24] defined algorithms for generating a minimum set of mutants based on the notion of dynamic subsumption. Their results confirmed the findings of Kintis *et al.* by demonstrating that the majority of mutants used by the MuJava and Proteum mutation testing tools are redundant. Later, Kurtz *et al.* [50] used static analysis techniques, such as symbolic execution to identify the minimum set of mutants.

Recently Papadakis *et al.* [26] used compilers to eliminate duplicated mutants, a special form of mutant redundancy. Duplicated mutants are those that are mutually equivalent but differ from the original program. In the study of Papadakis *et al.* [26] it is reported that 21% of all mutants is duplicated and can be removed by using compiler optimization techniques.

All the approaches discussed in this section either identified the problem of trivial/redundant mutants or used some form of redundancy elimination. However, none of them studied the differences of the commonly used operators from those suggested by the literature. Additionally, none of them uses disjoint mutants on real-world programs.

VII. Conclusions

This paper investigates the extent to which mutants used by popular mutation testing tools like PIT conform to mutation testing standards. Our study revealed a large divergence in the effectiveness of the popular mutants from the comprehensive ones. Comprehensive mutants are not only harder to kill but also score considerably higher than the common ones in most of the examined cases. Additionally, we report results by considering both concepts of disjoint and mutant easiness. Thus, we point out the importance of the problems introduced by both trivial and redundant mutants to be considered in future evaluations.

References

[1] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, 1997. [Online]. Available: http://doi.acm.org/10.1145/267580.267590

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

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

[4] 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.

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

[6] 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.

[7] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An Experimental Evaluation of Data Flow and Mutation Testing," *Software Pract. Exper.*, vol. 26, no. 2, pp. 165–176, 1996.

[8] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Software Engineering, IEEE Transactions on*, vol. 37, no. 5, pp. 649 –678, sept.-oct. 2011.

[9] 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.

[10] 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. [Online]. Available: http://dx.doi.org/10.1002/spe.2312

[11] H. Coles, "PIT." [Online]. Available: http://pitest.org/

[12] A. S. Namin and S. Kakarla, "The use of mutation in testing experiments and its sensitivity to external threats," in *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, 2011, pp. 342–352. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001461

[13] Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 214–224. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786858

[14] G. Denaro, A. Margara, M. Pezzè, and M. Vivanti, "Dynamic data flow testing of object oriented systems," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 947–958. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2015.104

[15] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 435–445. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568271

[16] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 246–256. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635921

[17] 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

[18] M. Papadakis and Y. L. Traon, "Metallaxis-fl: mutation-based fault localization," *Softw. Test., Verif. Reliab.*, vol. 25, no. 5-7, pp. 605–628, 2015. [Online]. Available: http://dx.doi.org/10.1002/stvr.1509

[19] T. A. Budd and D. Angluin, "Two Notions of Correctness and Their Relation to Testing," *Acta Informatica*, vol. 18, no. 1, pp. 31–45, 1982.

[20] 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. [Online]. Available: http://dx.doi.org/10.1002/stvr.242

[21] K.-C. Tai, "Theory of Fault-based Predicate Testing for Computer Programs," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 552–562, August 1996.

[22] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*, 2010, pp. 121–130. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2010.38

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

[24] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, 2014, pp. 21–30. [Online]. Available: http://dx.doi.org/10.1109/ICST.2014.13

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

[26] 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 *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 936–946. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2015.103

[27] A. J. Offutt and J. H. Hayes, "A semantic model of program faults," in *ISSTA*, 1996, pp. 195–200. [Online]. Available: http://doi.acm.org/10.1145/229000.226317

[28] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for java," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, 2009, pp. 297–298. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595750

[29] "Javancss - a source measurement suite for java." [Online]. Available: http://www.kclee.de/clemens/java/javancss/

[30] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Do redundant mutants affect the effectiveness and efficiency of mutation analysis?" in *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, 2012, pp. 720–725. [Online]. Available: http://dx.doi.org/10.1109/ICST.2012.162

[31] "The R project for statistical computing." [Online]. Available: https://www.r-project.org/

[32] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978. [Online]. Available: http://dx.doi.org/10.1109/C-M.1978.218136

[33] 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. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2011.93

[34] M. Nica, S. Nica, and F. Wotawa, "On the use of mutations and testing for debugging," *Softw., Pract. Exper.*, vol. 43, no. 9, pp. 1121–1142, 2013. [Online]. Available: http://dx.doi.org/10.1002/spe.1142

[35] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "Regression mutation testing," in *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, 2012, pp. 331–341. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336793

[36] B. K. Aichernig, J. Auer, E. Jöbstl, R. Korosec, W. Krenn, R. Schlick, and B. V. Schmidt, "Model-based mutation testing of an industrial measurement device," in *Tests and Proofs - 8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK, July 24-25, 2014. Proceedings*, 2014, pp. 1–19. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-09099-3_1

[37] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon, "Assessing software product line testing via model-based mutation: An application to similarity testing," in *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*, 2013, pp. 188–197. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2013.30

[38] M. Papadakis, C. Henard, and Y. L. Traon, "Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing," in *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, 2014, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/ICST.2014.11

[39] R. M. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Softw. Test., Verif. Reliab.*, vol. 9, no. 4, pp. 233–262, 1999. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1099-1689(199912)9:4⟨233::AID-STVR191⟩3.0.CO;2-3

[40] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Trans. Software Eng.*, vol. 40, no. 1, pp. 23–42, 2014. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2013.44

[41] M. Kintis, M. Papadakis, and N. Malevris, "Employing second-order mutation for isolating first-order equivalent mutants," *Softw. Test., Verif. Reliab.*, vol. 25, no. 5-7, pp. 508–535, 2015. [Online]. Available: http://dx.doi.org/10.1002/stvr.1529

[42] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. L. Traon, and J. Marion, "Sound and quasi-complete detection of infeasible test requirements," in *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, 2015, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/ICST.2015.7102607

[43] A. J. Offutt, "The Coupling Effect: Fact or Fiction," *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 131–140, December 1989.

[44] M. Papadakis and N. Malevris, “An empirical evaluation of the first and second order mutation testing strategies,” in *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings*, 2010, pp. 90–99. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2010.50

[45] W. E. Wong and A. P. Mathur, “Reducing the Cost of Mutation Testing: An Empirical Study,” *J. Syst. Software*, vol. 31, no. 3, pp. 185–196, December 1995.

[46] K. Tai, “Predicate-based test generation for computer programs,” in *Proceedings of the 15th International Conference on Software Engineering, Baltimore, Maryland, USA, May 17-21, 1993.*, 1993, pp. 267–276. [Online]. Available: http://portal.acm.org/citation.cfm?id=257572.257631

[47] G. Kaminski, P. Ammann, and J. Offutt, “Improving logic-based testing,” *Journal of Systems and Software*, vol. 86, no. 8, pp. 2002–2012, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2012.08.024

[48] ——, “Better predicate testing,” in *Proceedings of the 6th International Workshop on Automation of Software Test, AST 2011, Waikiki, Honolulu, HI, USA, May 23-24, 2011*, 2011, pp. 57–63. [Online]. Available: http://doi.acm.org/10.1145/1982595.1982608

[49] M. Papadakis and N. Malevris, “Mutation based test case generation via a path selection strategy,” *Information & Software Technology*, vol. 54, no. 9, pp. 915–932, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2012.02.004

[50] B. Kurtz, P. Ammann, and J. Offutt, “Static analysis of mutant subsumption,” in *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, 2015, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2015.7107454