2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation

Analysing and Comparing the Effectiveness of Mutation Testing Tools: A Manual Study

Marinos Kintis*, Mike Papadakis†, Andreas Papadopoulos*, Evangelos Valvis* and Nicos Malevris*
*Department of Informatics, Athens University of Economics and Business, Greece
†Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
kintism@aueb.gr, michail.papadakis@uni.lu, p3100148@dias.aueb.gr, p3130019@dias.aueb.gr, ngm@aueb.gr

***Abstract*—Mutation testing is considered as one of the most powerful testing methods. It operates by asking testers to design tests that reveal a set of mutants, which are purpose-made injected defects. Evidently, the strength of the method strongly depends on the used mutants. However, this dependence raises concerns regarding the mutation testing practice that is implemented by existing tools. Thus, it is probable that implementation inadequacies can lead to incompetent results. In this paper, we cross-evaluate three popular mutation testing tools for Java, namely MUJAVA, MAJOR and PIT, with respect to their effectiveness. We perform an empirical study of 3,324 manually analysed mutants from real-world projects and we find that there are large differences between the tools' effectiveness, ranging from 76% to 88%, with MUJAVA achieving the best results. We also demonstrate that no tool is able to subsume the others and provide practical recommendations on how to strengthen each one of the studied tools. Finally, our analysis shows that 11%, 12% and 7% of the mutants generated by MUJAVA, MAJOR and PIT are equivalent, respectively.**

I. INTRODUCTION

Undoubtedly, software testing is the most widely used practice for identifying software defects [1]. A typical testing scenario involves the generation of test cases based on which the actual behaviour of the program under analysis is evaluated. To guide the test generation process several techniques have been proposed. These techniques aim at specifying how to generate test cases that are either good at revealing faults or capable of providing confidence in the function of the system.

Among the several testing techniques, mutation analysis has shown to be quite powerful, capable of subsuming, or probably subsuming, almost all the structural testing techniques [1], [2]. Mutation asks testers to design test cases that reveal purpose-made injected defects. The underlying idea of mutation is that test cases capable of distinguishing the behaviour of the original program version from that of the versions with the injected defects are also capable of revealing the faulty behaviour of the program under test.

The program versions with the injected defects are called *mutants*. Clearly, mutation-testing effectiveness depends strongly on the mutants that are actually used [1]. Therefore, testers performing mutation testing should be cautious about the mutants they use. Recent research has shown that different sets of mutants can lead to different results when judging the effectiveness of the same test suites [3] and hence pointing out a threat to validity of the mutation-based testing studies. Similarly, the use of different mutation testing tools can lead to different results, for the same test suites, due to the different implementations and limitations of the tools.

To date, the theory and practice of mutation testing are mature enough and have led to practical testing tools [4], [5], which are widely used (mainly in research) [3]. However, a key question that remains is how well the various mutation-testing tools' implementations adhere to the theory of mutation and how well they perform the tasks that they were developed for. Investigating this issue forms the primary aim of this paper. Thus, we seek to investigate the effectiveness differences among popular mutation testing tools with the goal of identifying their weaknesses and strengths when used for generating effective test cases. Our aim is threefold: a) to inform practitioners about the effectiveness and relative cost of the studied mutation testing tools, b) to provide constructive feedback to tool developers on how to improve their tools, and c) to make researchers aware of the tools' inadequacies.

Our human analysis and comparison of three widely-used mutation testing tools for Java, namely MUJAVA, MAJOR and PIT, demonstrates that none of them subsumes the others. This fact indicates that all the studied tools can be strengthened by extending the set of mutants they support. Additionally, our results show that there are large differences between the effectiveness of the tools. In particular, according to a reference effectiveness measure, we found that MUJAVA scores best with 88%, followed by MAJOR with 80% and PIT with 76%. These results suggest that existing tools have a much lower effectiveness than what they should or what researchers believe they ought to. Therefore, our findings emphasise the need to build a reference mutation testing tool that will be strong enough and capable of at least subsuming the existing mutation testing tools.

Apart from the effectiveness of the tools, another concern, when using mutation, is its application cost. This is mainly due to the manual effort involved in constructing test cases and due to the effort needed for deciding when to stop testing. The former concern regards the need for generating test cases or test oracles while the latter pertains to the identification of the so-called *equivalent mutants*, i.e., mutants that are functionally equivalent to the original program. Both these tasks are labour-intensive and are performed manually. Our study shows that MUJAVA leads to 138 tests, MAJOR to 97 and PIT to 80. With respect to the number of equivalent mutants, MUJAVA, MAJOR and PIT produced 203, 94 and 43, respectively.

978-1-5090-3848-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SCAM.2016.28

The contributions of the present paper can be summarised in the following points:

1) An extensive, manual study of more than 3,000 mutants investigating the strengths and weaknesses of three, widely-used mutation testing frameworks for the Java programming language.
2) Insights on the relative cost of the tools' application in terms of the number of equivalent mutants that have to be manually analysed and the number of test cases that have to be generated.
3) Recommendations on specific mutation operators that need to be implemented in these frameworks in order to improve their effectiveness.
4) The public release of all the data of this study with the aim of enabling replication and future mutation testing experiments.

The rest of the paper is organised as follows: Section II presents the necessary background information and Section III outlines our study's motivation. In Section IV, we present the posed research questions and the adopted experimental procedure and, in Section V, we describe the obtained results. In Section VI, we discuss potential threats to the validity of this study, along with mitigating actions and in Section VII, previous research studies. Finally, Section VIII concludes this paper, summarising the key findings.

II. Background

This section details mutation testing and presents the studied mutation testing tools.

A. Mutation Testing

Applying mutation testing requires the generation and execution of a set of mutants with the candidate test cases. Mutants are produced using a set of syntactic rules called *mutation operators*. The process requires practitioners to design test cases that are able to distinguish the mutants' behaviour from that of the program under test, termed *original program* in mutation's terminology. In essence, these test cases should force the original program and its mutants to result in different outputs, i.e., they should *kill* its mutants.

Normally, the ratio of the killed mutants to the generated ones is an effectiveness measure that should quantify the ability of the test cases to reveal the system's defects. Unfortunately, among the killable mutants, there are some that cannot be killed, termed *equivalent mutants*. Equivalent mutants are syntactically different versions of the program under test, but semantically equivalent [6], [7]. These mutants must be discarded in order to have an accurate effectiveness measure, which is called *Mutation score*, i.e., the ratio of killed mutants to the number of killable mutants, and to decide when to stop testing. The problem of identifying and removing equivalent mutants is known as the *Equivalent Mutant Problem* [6], [7].

Regrettably, the Equivalent Mutant Problem has been shown to be undecidable in its general form [8], thus, no complete, fully automated solution can be devised to tackle it. This problem is largely considered an open issue in mutation's literature, but recent advances provide promising results towards practical, automated solutions, albeit partial, e.g., [6], [9].

Another problem of mutation testing is that it produces many mutants that are redundant, i.e., they are killed when other mutants are killed. These mutants can inflate the mutation score making it skew. Thus, previous research has shown that these mutants can have harmful effects on the mutation score measurement with the effect of leading experiments to incorrect conclusions [3]. Therefore, when mutation testing is used as a comparison basis, there is a need to deflate the mutation score measurement. This can be done by using the subset of subsuming mutants [3], [10] or disjoint mutants [11]. Disjoint mutants approximate the minimum "subset of mutants that need to be killed in order to reciprocally kill the original set" [11]. We utilise the term *disjoint mutation score* for the ratio of the disjoint mutants that are killed by the test cases under assessment (which in our case are those that were designed to kill the studied tools' mutants).

Mutation's effectiveness depends largely on the mutants that are used [1]. Thus, the actual implementation of mutation testing tools can impact the effectiveness of the technique. Indeed, many different mutation testing tools exist that are based on different architectural designs and implementations. As a consequence, it is not possible for researchers, and practitioners alike, to make an informed decision on which tool to use and on the strengths and weaknesses of the tools.

This paper addresses the aforementioned issue by analysing the effectiveness of three widely-used mutation testing tools for the Java programming language, namely MUJAVA, MAJOR and PIT, based on the results of an extensive manual study. Before presenting the conducted empirical study, the considered tools and their implementation details are introduced.

B. Selected tools

Mutation is popular [3] and, thus, many mutation testing tools exist. In this study we choose to work in Java since it is widely used by practitioners and forms the subject of most of the recent research papers. To select our subject tools, we performed a mini literature survey on the papers published during the last two years in the three leading Software Engineering conferences (ISSTA, (ESEC)FSE and ICSE) and identified the mutation testing tools that were used. The analysis resulted in three tools, MUJAVA [12], MAJOR [13] and PIT [14].

1) MUJAVA – Source Code Manipulation: MUJAVA [12] is one of the oldest Java mutation testing tools and has been used in many mutation testing studies. It works by directly manipulating the source code of the program under test and supports both method-level and class-level mutation operators. The former handle primitive features of programming languages, such as arithmetic operators, whereas the latter handle object-oriented features, such as inheritance. Note that MUJAVA adopts the selective mutation approach [15], i.e., it implements a set of 5 operators whose mutants subsume the mutants generated by other mutation operators not included in

TABLE I
MUTATION OPERATORS OF MUJAVA.

Mutation Operator	**Description**
AORB: *Arithmetic Operator Replacement Binary*	$\{(op_1, op_2) \mid op_1, op_2 \in \{+, -, \star, /, \%\} \wedge op_1 \neq op_2\}$
AORS: *Arithmetic Operator Replacement Short-Cut*	$\{(op_1, op_2) \mid op_1, op_2 \in \{++, --\} \wedge op_1 \neq op_2\}$
AOIU: *Arithmetic Operator Insertion Unary*	$\{(v, -v)\}$
AOIS: *Arithmetic Operator Insertion Short-cut*	$\{(v, --v), (v, v--), (v, ++v), (v, v++)\}$
AODU: *Arithmetic Operator Deletion Unary*	$\{(+v, v), (-v, v)\}$
AODS: *Arithmetic Operator Deletion Short-cut*	$\{(--v, v), (v--, v), (++v, v), (v++, v)\}$
ROR: *Relational Operator Replacement*	$\{((\mathrm{a}\ op\ \mathrm{b}),\ \texttt{false}), ((\mathrm{a}\ op\ \mathrm{b}),\ \texttt{true}), (op_1,\ op_2) \mid op_1, op_2 \in \{>, >=, <, <=, ==, !=\} \wedge op_1 \neq op_2\}$
COR: *Conditional Operator Replacement*	$\{(op_1, op_2) \mid op_1, op_2 \in \{\texttt{\&\&}, \vert\vert, \texttt{\textasciicircum}\} \wedge op_1 \neq op_2\}$
COD: *Conditional Operator Deletion*	$\{(!cond, cond)\}$
COI: *Conditional Operator Insertion*	$\{(cond, !cond)\}$
SOR: *Shift Operator Replacement*	$\{(op_1, op_2) \mid op_1, op_2 \in \{>>, >>>, <<\} \wedge op_1 \neq op_2\}$
LOR: *Logical Operator Replacement*	$\{(op_1, op_2) \mid op_1, op_2 \in \{\texttt{\&}, \vert, \texttt{\textasciicircum}\} \wedge op_1 \neq op_2\}$
LOI: *Logical Operator Insertion*	$\{(v, \sim v)\}$
LOD: *Logical Operator Deletion*	$\{(\sim v, v)\}$
ASRS: *Short-Cut Assignment Operator Replacement*	$\{(op_1, op_2) \mid op_1, op_2 \in \{+=, -=, \star=, /=, \%=, \texttt{\&}=, \vert=, \texttt{\textasciicircum}=, >>=, >>>=, <<=\} \wedge op_1 \neq op_2\}$

TABLE II
MUTATION OPERATORS OF PIT.

Mutation Operator	**Description**
AP: *Argument Propagation*	$\{(nonVoidMethodCall(...,\ par),\ par)\}$
CB: *Conditionals Boundary*	$\{(op_1, op_2) \mid (op_1, op_2) \in \{(<, <=), (<=, <), (>, >=), (>=, >)\}\}$
CC: *Constructor Calls*	$\{(\texttt{new}\ AClass(), \texttt{null})\}$
I: *Increments*	$\{(op_1, op_2) \mid op_1, op_2 \in \{++, --\} \wedge op_1 \neq op_2\}$
IC: *Inline Constant*	$\{(c_1, c_2) \mid (c_1, c_2) \in \{(1, 0), ((\texttt{int})\ x,\ \texttt{x+1}), (1.0, 0.0), (2.0, 0.0), ((\texttt{float})\ x,\ 1.0), (\texttt{true},\ \texttt{false}), (\texttt{false}, \texttt{true})\}\}$
IN: *Invert Negatives*	$\{(v, -v)\}$
M: *Math*	$\{(op_1, op_2) \mid (op_1, op_2) \in \{(+, -), (-, +), (\star, /), (/, \star), (\%, \star), (\texttt{\&}, \vert), (\vert, \texttt{\&}), (\texttt{\textasciicircum}, \texttt{\&}), (<<, >>), (>>, <<), (>>>, <<)\}\}$
MV: *Member Variable*	$\{(\texttt{member\_var=...,member\_var=b}) \mid b \in \{\texttt{false}, 0, 0.0, \texttt{'\textbackslash u0000'}, \texttt{null}\}\}$
NC: *Negate Conditionals*	$\{(op_1, op_2) \mid (op_1, op_2) \in \{(==, !=), (!=, ==), (<=, >), (>=, <), (<, >=), (>, <=)\}\}$
NVMC: *Non Void Method Calls*	$\{(nonVoidMethodCall(),\ c) \mid c \in \{\texttt{false}, 0, 0.0, \texttt{'\textbackslash u0000'}, \texttt{null}\}\}$
RC: *Remove Conditionals*	Removes or negates a conditional statement to force or prevent the execution of the guarded statements, e.g. $\{((\mathrm{a}\ op\ \mathrm{b}),\ \texttt{true})$ *or* $((\texttt{LHS \&\& RHS}),\ \texttt{RHS})\}$
RI: *Remove Increments*	$\{(--v, v), (v--, v), (++v, v), (v++, v)\}$
RS: *Remove Switch*	Changes all labels of the `switch` to the default one
RV: *Return Values*	$\{(\texttt{return}\ a,\ \texttt{return}\ b) \mid (a, b) \in \{(\texttt{true}, \texttt{false}), (\texttt{false}, \texttt{true}), (0, 1), ((\texttt{int})\, x, 0), ((\texttt{long})\, \texttt{x}, \texttt{x+1}), ((\texttt{float})\, x, \texttt{-(x+1.0)}), (\texttt{NAN}, 0), (\texttt{non-null}, \texttt{null}), (\texttt{null}, \texttt{throw}\ RuntimeException)\}\}$
S: *Switch*	Replaces the `switch`'s labels with the default one and vice versa (only for the first label that differs)
VMC: *Void Method Calls*	$\{(voidMethodCall(), \varnothing)\}$

this set. Table I presents the method-level operators of the tool, along with a succinct description of the performed changes. For instance, AORB replaces binary arithmetic operators with each other and AODS deletes the ++ and −− arithmetic operators.

2) PIT – Bytecode Manipulation: PIT [14] is a mutation testing framework that targets primarily the industry but has also been used in many research studies. PIT works by manipulating the resulting bytecode of the program under test and employs mutation operators that affect primitive programming language features, similarly to the method-level operators of MUJAVA. Table II describes the corresponding operators. By comparing this table with Table I, it can be seen that PIT implements differently specific mutation operators of MUJAVA, for instance, the changes imposed by PIT's Conditionals Boundary (CB) operator are a subset of the ones of MUJAVA's Relational Operator Replacement (ROR). Additionally, it employs mutation operators that are not implemented in MUJAVA, e.g. the Void Method Calls (VMC) and Constructor Calls (CC) operators. Finally, it should be mentioned that since PIT's changes are performed at the bytecode level, they cannot always be mapped onto source code ones.

3) MAJOR – AST MANIPULATION: MAJOR [13] is a mutation testing framework whose architectural design places it between the aforementioned ones: it manipulates the abstract syntax tree (AST) of the program under test. MAJOR employs mutation operators that have similar scope to the previously-described ones, but utilises specialised versions of specific operators that are supposed to be as effective as their traditional counterparts [16], cf. MAJOR's and MUJAVA's ROR operators. The implemented mutation operators of the tool are based on selective mutation, similarly to MUJAVA. Table III summarises MAJOR's operators and their imposed changes. Compared to MUJAVA's operators, it is evident that the two tools share many mutation operators, but implement them differently. Compared to PIT, most operators of MAJOR impose a superset of changes with respect to the corresponding ones of PIT and there are operators of PIT that are completely absent from MAJOR.

III. MOTIVATION

Mutation testing is important since it is considered as one of the most effective testing techniques. Its fundamental premise, as coined by Geist et al. [17], is that:

TABLE III
MUTATION OPERATORS OF MAJOR.

Mutation Operator	Description
AOR: *Arithmetic Operator Replacement*	$\{(op_1, op_2) \mid op_1, op_2 \in \{\texttt{+}, \texttt{-}, \star, \texttt{/}, \texttt{\%}\} \wedge op_1 \neq op_2\}$
LOR: *Logical Operator Replacement*	$\{(op_1, op_2) \mid op_1, op_2 \in \{\texttt{\&}, \vert, \texttt{\^{}}\} \wedge op_1 \neq op_2\}$
COR: *Conditional Operator Replacement*	$\{(\texttt{\&\&}, op_1), (\vert\vert, op_2) \mid op_1 \in \{\texttt{==}, \mathtt{LHS}, \mathtt{RHS}, \mathbf{false}\}, op_2 \in \{\texttt{!=}, \mathtt{LHS}, \mathtt{RHS}, \mathbf{true}\}\}$
ROR: *Relational Operator Replacement*	$\{(>, op_1), (<, op_2), (>=, op_3), (<=, op_4), (==, op_5), (!=, op_6) \mid op_1 \in \{>=, !=, \mathbf{false}\}, op_2 \in \{<=, !=, \mathbf{false}\}, op_3 \in \{>, ==, \mathbf{true}\}, op_4 \in \{<, ==, \mathbf{true}\}, op_5 \in \{<=, >=, \mathbf{false}, \mathtt{LHS}, \mathtt{RHS}\}, op_6 \in \{<, >, \mathbf{true}, \mathtt{LHS}, \mathtt{RHS}\}\}$
SOR: *Shift Operator Replacement*	$\{(op_1, op_2) \mid op_1, op_2 \in \{>>, >>>, <<\} \wedge op_1 \neq op_2\}$
ORU: *Operator Replacement Unary*	$\{(op_1, op_2) \mid op_1, op_2 \in \{+, -, \sim\} \wedge op_1 \neq op_2\}$
STD: *Statement Deletion Operator*	$\{(--v, v), (v--, v), (++v, v), (v++, v), (aMethodCall(), \varnothing), (a\ op_1\ b, \varnothing) \mid op_1 \in \{+=, -=, \star=, /=, \%=, \&=, \vert=, \texttt{\^{}}=, >>=, >>>=, <<=\}\}$
LVR: *Literal Value Replacement*	$\{(c_1, c_2) \mid (c_1, c_2) \in \{(0, 1), (0, -1), (c_1, -c_1), (c_1, 0), (\mathbf{true}, \mathbf{false}), (\mathbf{false}, \mathbf{true})\}$

> *"If the software contains a fault, it is likely that there is a mutant that can only be killed by a test case that also reveals the fault."*

This premise has been empirically investigated by many research studies which have shown that mutation adequate test suites, i.e., test suites that kill all killable mutants, are more effective than the ones generated to cover various control and data flow coverage criteria [2]. Therefore, researchers use mutation testing as a way to either compare other test techniques or as a target to automate.

Overall, a recent study by Papadakis et al. [3] shows that mutation testing is popular and widely-used in research (probably due to its remarkable effectiveness). In view of this, it is mandatory to ensure that mutation testing tools are powerful and do not bias (due to implementation inadequacies or missing mutation operators) the existing research.

To reliably compare the selected tools, it is mandatory to account for mutant subsumption [3] when performing a complete testing process, i.e., using mutation-adequate tests. Accounting for mutant subsumption is necessary in order to avoid bias from subsumed mutants [3], while complete testing ensures the accurate estimation of the tools' effectiveness. An inaccurate estimation may happen when failing to kill some killable mutants, which consequently results in failing to design tests (to kill these mutants) and, thus, underestimate effectiveness. Even worse, the use of non-adequate test suites ignores hard to kill mutants which are important [18] and among those that (probably) contribute to the test process. Since we know that very few mutants contribute to the test process [3], the use of non-adequate test suites can result in major degradation of the measured effectiveness. For all these reasons, we use mutation adequate test suites specially designed for each tool that we study.

IV. EMPIRICAL STUDY

This section presents the settings of our study, by detailing the research questions, the followed procedure and the design of our experiment.

A. Research Questions

Mutation testing's aim is to help testers design high quality test suites. Therefore, the first question to ask is whether there is a tool that is more effective or at least as effective as the other tools. In other words, we want to measure how effective are the designed tests based on one tool in killing the mutants of the other tools. Hence we ask:

RQ1: *Does any mutation testing tool lead to tests that kill all the killable mutants produced by the other tools?*

This comparison enables checking whether there is a tool that is capable of subsuming the others, i.e., whether the mutation adequate tests of one tool can kill all the killable mutants of the others. A positive answer to the above question indicates that a single tool is superior to the others, in terms of effectiveness. A negative answer to this question indicates that the tools are generally incomparable, meaning that there are mutants not covered by the tools. We view these missed mutants as weaknesses of the tools.

To further compare mutation testing tools and identify their weaknesses we need to assess the quality of the test suites that they lead to. This requires either an independent, to the used mutants, effectiveness measure or a form of "ground truth", i.e., a golden set of mutants. Since both are not known, we constructed a reference mutant set, the set of disjoint mutants, from the superset of mutants produced by all the studied tools together and all generated test cases. We use the disjoint set of mutants to avoid inflating the reference set from the duplicated, i.e., mutants equivalent to each other but not to the original program [6], and redundant mutants, i.e., mutants subsumed by other mutants of the merged set of mutants [3]. Both duplicated and redundant mutants inflate the mutation score measurement with the unfortunate result of committing Type I errors [3]. Since in our case these types of mutants are expected to be numerous, as the tools support many common types of mutants, the use of disjoint mutants was imperative. Therefore we ask:

RQ2: *How do the studied tools perform compared to a reference mutant set?*

This comparison enables the ranking of the tools with respect to their effectiveness. The use of the reference mutant set also helps aggregate all the data and quantify the relative strengths and weaknesses of the studied tools in one measure (the disjoint mutation score). Given the effectiveness ranking offered by this comparison, a natural question to ask is:

RQ3: *Which is the relatively most effective tool to use?*

Identifying the most effective mutation testing tool and quantifying the effectiveness differences between the tools is

TABLE IV
TEST SUBJECT DETAILS: GENERATED MUTANTS, DISJOINT MUTANTS AND REFERENCE MUTANT SET.

			# Generated Mutants			# Disjoint Mutants			# Mutants
Test Subject	LoC	Method	MAJOR	PIT	MUJAVA	MAJOR	PIT	MUJAVA	Reference Mutant Set
Commons-Math	16,489	gcd	133	79	237	7	9	9	8
		orthogonal	120	65	155	11	11	11	11
Commons	17,294	toMap	23	50	32	6	6	5	7
		subarray	25	27	64	6	6	3	6
		lastIndexOf	29	43	81	11	8	13	13
		capitalize	37	42	69	5	4	4	6
		wrap	71	70	198	12	7	16	13
Pamvotis	5,505	addNode	89	53	318	16	16	29	29
		removeNode	18	29	55	7	6	6	7
Triangle	47	classify	139	94	354	31	16	31	31
XStream	15,048	decodeName	73	81	156	8	7	8	9
Bisect	37	sqrt	51	29	135	7	6	7	10
Total	**54,420**	-	**808**	**662**	**1,854**	**127**	**102**	**142**	**150**

important when choosing a tool to use but does not provide any constructive information on the weaknesses of the tools. Furthermore, this information fails to provide researchers and tool developers constructive feedback on how to build future tools or strengthen the existing ones. Therefore, we seek to analyse the observed weaknesses and ask:

RQ4: *Are there any actionable findings on how to improve the effectiveness of the studied tools?*

Our intentions thus far have been concentrated on the relative effectiveness of the tools. While this is important when using mutation, another major concern is the cost of its application. Mutation testing is considered to be expensive due to the manual effort involved in identifying equivalent mutants and designing test cases. Since we manually assess and apply the mutation testing practice of the studied tools we ask:

RQ5: *What is the relative cost, measured by the number of tests and number of equivalent mutants, of applying mutation testing with the studied tools?*

An answer to this question can provide useful information to both testers and researchers regarding the trade-offs between cost and effectiveness. Also, this analysis will better reflect the differences of the tools from the cost perspective.

B. Manual Study

In order to assess the effectiveness of the studied tools, we manually applied them to test parts of several real-world projects. Since manual analysis requires considerable resources, analysing a complete project is infeasible. Thus, we picked and analysed 12 methods from 6 Java test subjects for 3 independent times, once per studied tool. Thus, in total, we manually analysed 36 methods and 3,324 mutants which constitutes one of the largest studies in the literature of mutation testing, e.g., Yao et al. [19] consider 4,181 mutants, Baker and Habli [20] consider 2,555. Further, the present study is the only one in the literature to consider manually analysed mutants when comparing the effectiveness of different mutation testing tools (see also Section VII). The rest of this section discusses the test subjects, tool configuration and the manual procedure we followed in order to perform mutation testing.

1) Test Subjects: We selected 12 methods to perform our experiment; 10 of them were randomly picked from 4 real-world projects (Commons-Math, Commons, Pamvotis and XStream) and another 2 (Triangle and Bisect) from the mutation testing literature [1]. Details regarding the selected subjects are presented in Table IV. The table presents the name of the test subjects, their source code lines as reported by the `cloc` tool, the names of the studied methods, the number of generated and disjoint mutants per tool and the number of the resulting mutants of the reference mutant set.

2) Tool Configuration: We used the three mutation testing tools for Java that were mentioned in the papers published during the last two years in the ISSTA, (ESEC)FSE and ICSE conferences. Thus, we used version 3 of MUJAVA, version 1.1.8 of MAJOR and version 1.1.10 of PIT [14] and applied all the provided mutation operators. In the case of MUJAVA, only the method-level operators were employed, since the other tools do not provide object-oriented operators.

3) Manual Analysis Procedure: The primary objective of our experiment is to accurately measure the effectiveness of the studied tools. Thus, we performed complete manual analysis (by designing tests that kill all the killable mutants and manually identified the equivalent mutants) of the mutants produced by the selected tools. Performing this task is labour-intensive and error-prone. Thus, to avoid bias from the use of different tools we asked different users to perform mutation testing on our subjects. To find this number of qualified human subjects we turned to third- and fourth-year Computer Science students of the Department of Informatics at the Athens University of Economics and Business and adopted a two-phase manual analysis process:

- The selected methods were given to students attending the "Software Validation, Verification and Maintenance" course (Spring 2015 and Fall 2015), taught by Prof. Malevris, in order to analyse the mutants of the studied tools, as part of their coursework. The participating students were selected based on their overall performance and their grades at the programming courses. Additionally, they all attended an introductory lecture on mutation testing and appropriate tutorials before the beginning of their coursework. To facilitate the smooth completion of their projects and the correct application of mutation, the students were closely supervised, with regular team meetings throughout the semester.

- The designed test cases and detected equivalent mutants were manually analysed and carefully verified by at least one of the authors.

To generate the mutation adequate test suites, the students were first instructed to generate branch adequate test suites and then to randomly pick a live mutant and attempt to kill it based on the RIP Model [1]. Although the detection of killable mutants is an objective process, i.e., the produced test case either kills the corresponding mutant or not, the detection of equivalent ones is a subjective one. To deal with this issue, all students were familiarised with the RIP Model [1] and the sub-categories of equivalent mutants described by Yao et al. [19]. Also, all detected equivalent mutants were independently verified. To support replication and wider scrutiny of our study, we made all its publicly available [21].

C. Methodology

To answer the stated RQs, we applied mutation testing by independently using each one of the selected tools. Thus, we generated three different mutation adequate test sets per analysed method. Next, we minimised these test sets by checking, for each contained test case, whether its removal would result in a decreased mutation score [1]. Note that the removal of the redundant tests is necessary in order to produce an accurate disjoint mutant set. We used the resulting tests and computed the set of disjoint mutants produced by each one of the tools. We then constructed the reference mutant set by identifying the disjoint mutants (using all the produced tests) of the mutant set composed of all mutants of all the studied tools. To compute the disjoint mutant sets we extended PIT and MUJAVA to produce a matrix that records all test cases that kill a mutant. The disjoint set of mutants was computed using the "Subsuming Mutants Identification" process that is described in the study of Papadakis et al. [3]. Here, we use the term "disjoint" mutants, instead of "subsuming" ones since this was the original term that was used by the first study that introduced them and suggested their use as a metric that can accurately measure test effectiveness [11].

To answer RQ1, for each selected tool we used its mutation adequate test suite and calculated the mutation score and disjoint mutation score that it achieves when it is evaluated with the mutants produced by the other tools. This process can be viewed as an objective comparison between the tools, i.e., a comparison that evaluates how the tests designed for one tool perform when evaluated in terms of the other tool. In case the tests of one tool can kill all the mutants produced by the other tool, then this tool subsumes the other. Otherwise, the two tools are incomparable.

To answer RQ2, we used the tests that were specifically designed for each one of the studied tools and measured the score they achieve when evaluated against the reference mutant set. This score provides the common ground to compare the tools and rank them with respect to their effectiveness and, thus, also answer RQ3.

To answer RQ4, for each tool we manually analysed the mutants that were not killed by the tests of the other tools with the intention of identifying inadequacies in the tools' mutant sets. We then gathered all these instances and identified how we could complement each one of the tools in order to improve its effectiveness and reach the level of the reference mutant set. Finally, to answer RQ5, we measured and report the number of tests and equivalent mutants that we found. We also analysed and measured the number of equivalent mutants that are common (and non-common) between each pair of the studied tools.

V. EMPIRICAL FINDINGS

This section presents the empirical findings of our study per posed research question.

A. RQ1: Tool's Cross-evaluation

This question investigates whether one of the studied tools subsumes the others. Table V presents the respective results. The table is divided into three parts (columns MAJOR, PIT, MUJAVA) and each of these parts is divided into two columns that correspond to the mutation adequate test sets of the remaining tools. Finally, each of these columns is split into two sub-columns that depict the mutation scores achieved by the corresponding test suites when considering all generated mutants (column "All") and the disjoint ones (column "Dis.").

By examining Table V, it becomes evident that none of the tools subsumes the others; all generated test suites face effectiveness losses when evaluated against the mutants of the other tools. Specifically, PIT's mutation adequate test suites perform the worst, with an effectiveness of approximately 95% with respect to MUJAVA and MAJOR when all mutants are considered; for the disjoint ones, this score drops to approximately 80% for MAJOR and 75% for MUJAVA. On the contrary, MUJAVA's mutation adequate test suites perform the best, with an effectiveness of 96% and 98% for MAJOR and PIT, when all mutants are considered and 91% and approximately 93% for the disjoint ones, respectively. MAJOR's results place it in the middle, with an effectiveness of approximately 99% and 97% for PIT and MUJAVA for all generated mutants; for the disjoint ones, its effectiveness drops to 96% and 85%, respectively.

B. RQ2: Comparison with Reference Mutation Tool

This question investigates how the tools' mutation adequate test suites fare against a reference mutation testing tool, simulated by the disjoint mutants of the union of all mutants of the studied tools. Figure 1 depicts the obtained findings. The figure presents the percentage of the mutants that can be killed by the corresponding mutation adequate test suites per method,

TABLE V
TOOLS' CROSS-EVALUATION RESULTS.

	MAJOR				PIT				MUJAVA			
	PIT-TS		MUJAVA-TS		MAJOR-TS		MUJAVA-TS		MAJOR-TS		PIT-TS	
Method	All	Dis.	All.	Dis.	All	Dis.	All.	Dis.	All	Dis.	All	Dis.
gcd	97.4%	71.4%	97.4%	71.4%	100.0%	100.0%	97.1%	81.8%	99.5%	88.9%	100.0%	100.0%
orthogonal	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
toMap	88.9%	66.7%	77.8%	83.3%	100.0%	100.0%	91.7%	83.3%	100.0%	100.0%	96.0%	80.0%
subarray	90.0%	66.7%	85.0%	50.0%	100.0%	100.0%	95.8%	83.3%	100.0%	100.0%	91.1%	66.7%
lastIndexOf	100.0%	100.0%	92.6%	91.0%	100.0%	100.0%	97.6%	87.5%	100.0%	100.0%	97.4%	85.0%
capitalize	93.5%	60.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	98.2%	75.0%
wrap	98.4%	91.7%	100.0%	100.0%	98.5%	85.7%	98.5%	85.7%	98.9%	93.8%	96.1%	81.2%
addNode	98.7%	93.8%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	76.5%	37.9%	78.2%	34.5%
removeNode	93.8%	85.7%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	91.7%	50.0%	93.8%	66.7%
classify	90.2%	61.3%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	98.1%	90.3%	94.9%	58.1%
decodeName	98.0%	87.5%	100.0%	100.0%	96.9%	71.4%	100.0%	100.0%	95.3%	62.5%	99.2%	87.5%
sqrt	93.6%	71.4%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	97.5%	71.4%
Average	**95.2%**	**79.7%**	**96.1%**	**91.3%**	**99.6%**	**96.4%**	**98.4%**	**93.5%**	**96.7%**	**85.3%**	**95.2%**	**75.5%**

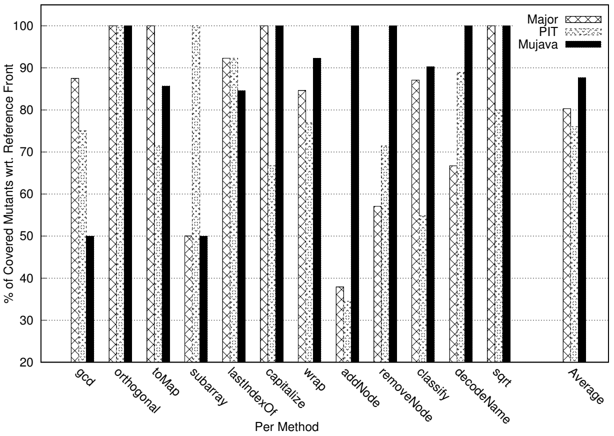

Fig. 1. Comparison of Mutation Adequate Test Suites against Reference Mutation Set.

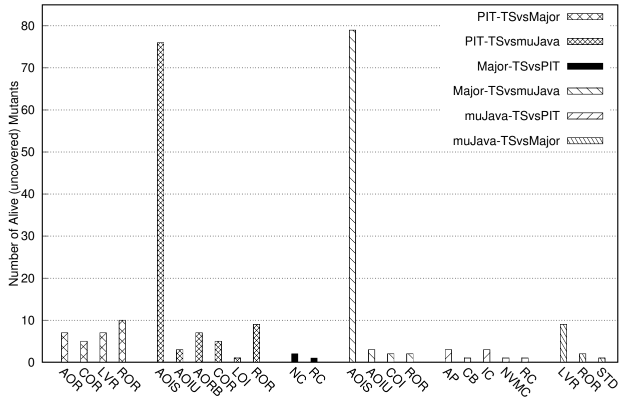

Fig. 2. Cross-Evaluation Experiment: Number of Alive Mutants per Mutation Operator, Test Suite and Tool.

along with the average score for all methods. Although, the performance of the tools varies depending on the considered method, it can be seen that, on average, MUJAVA realises an 88% effectiveness score, followed by MAJOR and PIT with 80% and 76%, respectively. An interesting observation from these results is that all tools have important inadequacies that range from 0-65%. On average, the differences are 12%, 20% and 24% for MUJAVA, MAJOR and PIT.

C. RQ3: Better Performing Tool

This question regards the identification of the most effective tool. From the results of the previous research question it should be evident that on average MUJAVA is the top ranked tool, followed by MAJOR and PIT. MUJAVA achieves a higher mutation score (w.r.t. the reference mutant set) than MAJOR in 5 cases, equal in 4 and lower in 3 cases. It is also the tool with the highest stability. Therefore, we conclude that according to our sample MUJAVA is the most effective tool.

D. RQ4: Tools' Weaknesses & Recommendations

This question is concerned with ways of improving the mutation testing practice of the studied tools. To this end, Figure 2 presents the mutants per tool (divided into mutation operators) that remained alive after the application of the mutation adequate test suites of the other tools. The figure is divided into six parts, each one illustrating the live mutants of a corresponding tool with respect to the mutation adequate test suite of another tool.

1) ***Recommendations:*** PIT*:* As can be seen from Figure 2, PIT's mutation testing practice can be improved by strengthening its mutation operators that deal with relational and conditional expressions, e.g. CB, NC, RC and by incorporating the changes of the COR and ROR operators of MUJAVA and MAJOR. Additionally, PIT's Math operator can be enhanced by including the changes of MUJAVA's AORB and MAJOR's AOR operators. Finally, from the figure, it is apparent that PIT's tests fail to kill mutants generated by MUJAVA's AOIS mutation operator, thus, the tool must implement a better version of its Increments operator to include the addition of the pre- and post-increment and decrement arithmetic operators.

*2) **Recommendations:** Major:* Similarly to PIT, Major's tests fail to cover muJava's AOIS mutants. Thus, the tool's practice can be enhanced by implementing the AOIS operator. Additionally, Major will benefit by adding an operator that negates arithmetic variables, analogous to muJava's AOIU. Finally, we observed that the tests of Major failed to kill some mutants generated by muJava's ROR operator. Recall that Major implements a specialised version of ROR that induces only a subset of its changes. The live mutants indicate a weakness in this specialised set that can lead to test effectiveness loss. An example of such a weakness manifested at line 161 of the `decodeName` method where muJava's ROR changed the sub-expression `c == escapeReplacementFirstChar` to `c > escapeReplacementFirstChar`. It is noted that the issue with the ROR mutants is not a defect of Major but an implementation choice. Indeed, Major was built based on the studies of Kaminski et al. [23] and Just and Schweiggert [16], which suggest that LOR and ROR operators produce many redundant mutants that can be reduced by using the respective rules shown in Table III. However, both of the above studies were based on weak mutation [1]. This practice as pointed out by Papadakis and Malevris [22] "does not hold for strong mutation because of the existence of strongly equivalent mutants" [22]. Along the same lines, the work of Lindström and Márki [24] provided empirical evidence supporting the above argument, which also confirms our results.

*3) **Recommendations:** muJava:* As can be seen from Figure 2, muJava's weaknesses centre around mutation operators that affect literal values, namely Major's LVR and PIT's IC. Thus, muJava will benefit by implementing such operators. Furthermore, we found muJava's implementation of the ROR mutation operator inconsistent; for example, at line 25 of the `wrap` method, the tool did not replace the original statement, `if (newLineStr == null)`, with `if (true)`, as it was supposed to, leading to inadequacies in the resulting test suites. Similar examples are present at line 248 of `toMap` and 1282 of `lastIndexOf`. These implementation defects lower the test effectiveness of the resulting mutation adequate test suites and addressing them will improve the tool's test quality.

E. RQ5: Tools' Application Cost

The answer to this question provides insights on the relative cost of the studied tools' application in terms of the number of equivalent mutants that have to be manually analysed and the number of needed test cases. Table VI presents the corresponding findings. The table is divided into three parts, each one for a studied tool, and presents the examined cost metrics in the sub-columns of these parts ("#Eq." and "#Tests").

We can observe that 12% of Major's mutants are equivalent, 6% of PIT's and 11% of muJava's ones. Thus, PIT requires the least amount of human effort in identifying the generated equivalent mutants, whereas muJava the highest. Regarding the number of killing test cases the tools require, Major requires 97 test cases, PIT 80 and muJava 138. Again, PIT requires the least amount of effort in generating mutation adequate test suites.

TABLE VI
Manual Analysis Results: #Equivalent Mutants & #Tests.

	Major		PIT		muJava	
Method	#Eq.	#Tests	#Eq.	#Tests	#Eq.	#Tests
gcd	17	6	9	7	23	7
orthogonal	3	8	0	8	5	9
toMap	5	7	2	5	7	5
subarray	5	6	3	4	8	6
lastIndexOf	2	8	1	7	4	12
capitalize	6	5	1	6	14	9
wrap	8	10	4	6	19	7
addNode	11	8	3	8	33	34
removeNode	2	5	0	3	7	6
classify	7	25	1	16	38	27
decodeName	24	5	16	6	28	10
sqrt	4	4	3	4	17	6
Total	**94**	**97**	**43**	**80**	**203**	**138**

The previously-described results indicate that PIT is the most efficient tool and muJava the most expensive one, with Major standing in the middle. Considering that PIT was found the least effective tool, it is no surprise that it is the most efficient one. Analogously, muJava requires the most effort, a fact justified by its high effectiveness. Major's practice is placed in the middle, presenting a compromise between effectiveness and efficiency.

To better understand the nature of the generated equivalent mutants, Figure 3 illustrates the contribution of each mutation operator to the generated killable and equivalent mutants per tool. In the case of muJava, AOIS and ROR generate most of the tool's equivalent mutants. For Major, ROR generates most of the equivalent mutants, followed by LVR, AOR and COR. In the case of PIT, RC and CB generate the most equivalent mutants.

Finally, we also examined the percentage of common equivalent mutants among the studied tools. The corresponding results are depicted in Table VII. Overall, we observed that the tools generate different equivalent mutants, with PIT and Major generating the most common (17%) and PIT and muJava the least (6.6%). When all tools' equivalent mutants are considered, this percentage drops to 4.1%. This observation indicates that the application of more than one mutation testing frameworks will not benefit from the already analysed mutants of the one tool.

VI. Threats to Validity

As every empirical study, this one faces specific threats to its validity. Here we discuss these threats along with the actions we took to mitigate them.

External Validity. External validity refers to the ability of a study's results to generalise. To mitigate the underlying threats, we utilised 6 widely-used test subjects and manually analysed 12 methods whose application domain varies. Although we

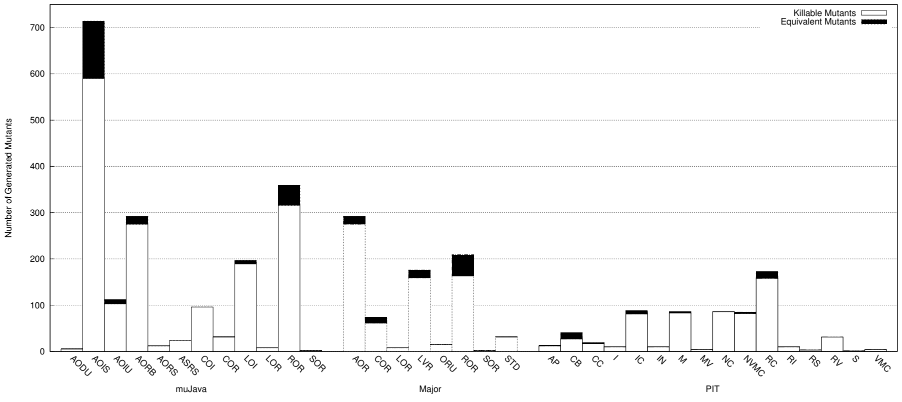

Fig. 3. Contribution of Mutation Operators to Generated and Equivalent Mutants per Tool.

TABLE VII
PERCENTAGE OF COMMON EQUIVALENT MUTANTS AMONG THE TOOLS.

Method	MAJOR-PIT	MUJAVA-MAJOR	MUJAVA-PIT	All
gcd	19.2%	17.5%	12.5%	8.2%
orthogonal	0.0%	12.5%	0.0%	0.0%
toMap	14.3%	25.0%	11.1%	7.1%
subarray	37.5%	15.4%	18.2%	12.5%
lastIndexOf	33.3%	0.0%	0.0%	0.0%
capitalize	14.3%	15.0%	0.0%	0.0%
wrap	25.0%	18.5%	8.7%	6.5%
addNode	14.3%	18.2%	5.6%	4.3%
removeNode	0.0%	11.1%	0.0%	0.0%
triangle	0.0%	6.7%	0.0%	0.0%
xstream	17.5%	32.7%	13.6%	5.9%
bisect	28.6%	9.5%	10.0%	4.2%
Average	**17.0%**	**15.2%**	**6.6%**	**4.1%**

cannot claim that our results are completely generalisable, our findings indicate specific inadequacies in the mutants produced by the studied tools. These involve incorrect implementation or not supported mutation operators, evidence that is almost impossible to be case-specific.

Internal Validity. Internal validity includes potential threats to the conclusions we drew. Our conclusions are based on the manual analysis, i.e., on the identified equivalent mutants and mutation adequate test suites. Thus, our analysis is a potential source of threats. To control this fact, we ensured that this analysis was performed by different persons to avoid any bias in the results and that all results produced by students were independently checked for correctness by at least one of the authors. Another potential threat is due to the fact that we did not control the test suite size. However, our study focuses on investigating the effectiveness of the studied tools when used as a means to generate strong tests [22]. To cater for wider scrutiny, we made publicly available all the data of this study [21].

Construct Validity. Construct validity pertains to the appropriateness of the measures utilised in our experiments. For the effectiveness comparison, we used the mutation score and disjoint mutation score measurements. These are well-established measures in mutation testing literature [3]. Another threat originates from evaluating the tools' effectiveness based on the reference mutant set. We deemed this particular measure appropriate because it constitutes a metric that combines the overall performance of the tools and enables their ranking. Finally, the number of equivalent mutants and generated tests might not reflect the actual cost of applying mutation. We adopted these metrics because they involve manual analysis which is a dominant cost factor when testing.

VII. RELATED WORK

Mutation testing is a well-studied technique with a rich history of publications, as recorded in the surveys of Offutt [25] and Yia and Harman [2].

Manual analysis has been used extensively in the mutation testing literature. Yao et al. [19] analysed 4,181 mutants to provide insights into the nature of equivalent and stubborn mutants. Nan et al. [26] manually analysed 2,919 mutants to compare test cases generated for mutation testing with the ones generated for various control and data flow coverage criteria. Deng et al. [27] analysed 5,807 mutants generated by MUJAVA to investigate the effectiveness of the SDL mutation operator.

Previous work on the differences of mutation testing frameworks for Java is due to Delahaye and Du Bousquet [4]. Delahaye and Du Bousquet compare several tools based on various criteria, such as the supported mutation operators, implementation differences and ease of usage. The study concluded that different mutation testing tools are appropriate to different scenarios.

A similar study was performed by Rani et al. [28]. This study compared several Java mutation testing tools based on a set of manually generated test cases. The authors concluded that PIT generated the smallest number of mutants, most of which were killed by the employed test suite (only 2% survived), whereas, MUJAVA generated the largest number of mutants, 30% of which survived.

Gopinath et al. [5] investigated the effectiveness of mutation testing tools by using various metrics, e.g., comparing the mutation score (obtained by the test subjects' accompanying test suites) and number of disjoint/minimal mutants that they produce. They found that the examined tools exhibit considerable variation of their performance and that no single tool is consistently better than the others.

The main difference between our study and the aforementioned ones is that we manually analysed the tools' mutants by performing complete analysis. This is a mandatory requirement (see Section III) for performing a reliable effectiveness comparison between the tools. This constitutes the strength of the present paper as it is the first one in the literature to perform manual analysis in comparing mutation tools. Further, we identified specific limitations of the tools and provided actionable recommendations on how each of the tools can be improved. Lastly, we analysed and reported the number and characteristics of equivalent mutants produced by each tool.

VIII. Conclusions & Future Work

Researchers largely base their findings on the results provided by mutation testing tools. This practice increases the need for reliable, effective and robust tools. While most of the existing tools have been shown to be robust [4], their effectiveness remains unexplored.

We manually analysed and investigated the use of three popular Java mutation testing tools with the intention of making practitioners and researchers aware of their relative cost, effectiveness and limitations. Our analysis revealed large differences among the tools, with respect to both cost and effectiveness. In particular, it showed that muJava is the most effective tool with 88% effectiveness score, followed by Major and PIT with 80% and 76%, respectively. We also identified several ways of strengthening each of the tools so that their effectiveness becomes at least as much as each of the others in turn.

Future work includes the investigation of the effectiveness of additional tools, e.g., version 4 of muJava, and the recently proposed improvements to PIT's practice [29]. We also plan to conduct further experiments with more test subjects.

Acknowledgment

Marinos Kintis and Nicos Malevris are partly supported by the Research Centre of Athens University of Economics and Business (RC AUEB).

References

[1] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. New York, NY, USA: Cambridge University Press, 2008.

[2] 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, 2011.

[3] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. L. Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the 2016 International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016.

[4] M. Delahaye and L. Du Bousquet, "Selecting a software engineering tool: lessons learnt from mutation analysis," *Software: Practice and Experience*, vol. 45, no. 7, pp. 875–891, 2015.

[5] 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.

[6] 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 International Conference on Software Engineering*, vol. 1, May 2015, pp. 936–946.

[7] M. Kintis, "Effective methods to tackle the equivalent mutant problem when testing software with mutation," Ph.D. dissertation, Department of Informatics, Athens University of Economics and Business, 2016.

[8] 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.

[9] M. Kintis and N. Malevris, "MEDIC: A static analysis framework for equivalent mutant identification," *Information and Software Technology*, vol. 68, pp. 1 – 17, 2015.

[10] 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.

[11] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *Proceedings of the 17th Asia-Pacific Software Engineering Conference*, November 2010, pp. 300–309.

[12] 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.

[13] R. Just, "The Major mutation framework: Efficient and scalable mutation analysis for Java," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2014, pp. 433–436.

[14] H. Coles. "the PIT mutation testing tool". "Last Accessed June 2016". [Online]. Available: "http://pitest.org/"

[15] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, pp. 99–118, Apr. 1996.

[16] R. Just and F. Schweiggert, "Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 490–507, 2015.

[17] R. Geist, A. Offutt, and J. Harris, F.C., "Estimation and enhancement of real-time software reliability through mutation analysis," *Computers, IEEE Transactions on*, vol. 41, no. 5, pp. 550–558, May 1992.

[18] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *Software Engineering, IEEE Transactions on*, vol. 32, no. 8, pp. 608–624, 2006.

[19] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Procs. of the 36th Inter. Conf. on Software Engineering*, 2014, pp. 919–930.

[20] R. Baker and I. Habli, "An empirical evaluation of mutation testing for improving the test quality of safety-critical software," *Software Engineering, IEEE Transactions on*, vol. 39, no. 6, pp. 787–805, 2013.

[21] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris. Supporting site for paper: Analysing and comparing the effectiveness of mutation testing tools: A manual study. [Online]. Available: http://pages.cs.aueb.gr/~kintism/papers/scam2016

[22] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *Software Reliability Engineering, 21st International Symposium on*, 2010, pp. 121–130.

[23] G. Kaminski, P. Ammann, and J. Offutt, "Improving logic-based testing," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2002–2012, 2013.

[24] B. Lindström and A. Márki, "On strong mutation and subsuming mutants," in *Procs. 11th Inter. Workshop on Mutation Analysis*, 2016.

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

[26] 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 *Software Testing, Verification and Validation Workshops, International Conference on*, April 2009, pp. 220–229.

[27] L. Deng, J. Offutt, and N. Li, "Empirical evaluation of the statement deletion mutation operator," in *Software Testing, Verification and Validation, IEEE Sixth International Conference on*, 2013, pp. 84–93.

[28] S. Rani, B. Suri, and S. K. Khatri, "Experimental comparison of automated mutation testing tools for Java," in *Reliability, Infocom Technologies and Optimization, 4th Inter. Conference on*, 2015, pp. 1–6.

[29] T. Laurent, A. Ventresque, M. Papadakis, C. Henard, and Y. L. Traon, "Assessing and improving the mutation testing practice of PIT," University of Luxembourg, Tech. Rep., 2016.