Mutation Testing Advances: An Analysis and Survey

Mike Papadakis[*], Marinos Kintis[*], Jie Zhang[‡], Yue Jia[†], Yves Le Traon[*], Mark Harman[†]
[*]Luxembourg University, Luxembourg, Luxembourg
[†]University College London, London, United Kingdom
[‡]Peking University, Beijing, China

Contents

Advances in Computers
ISSN 0065-2458
https://doi.org/10.1016/bs.adcom.2018.03.015

Abstract

Mutation testing realizes the idea of using artificial defects to support testing activities. Mutation is typically used as a way to evaluate the adequacy of test suites, to guide the generation of test cases, and to support experimentation. Mutation has reached a maturity phase and gradually gains popularity both in academia and in industry. This chapter presents a survey of recent advances, over the past decade, related to the fundamental problems of mutation testing and sets out the challenges and open problems for the future development of the method. It also collects advices on best practices related to the use of mutation in empirical studies of software testing. Thus, giving the reader a “mini-handbook”-style roadmap for the application of mutation testing as experimental methodology.

1. INTRODUCTION

How can we generate test cases that reveal faults? How confident are we with our test suite? Mutation analysis answers these questions by checking the ability of our tests to reveal some artificial defects. In case our tests fail to reveal the artificial defects, we should not be confident on our testing and should improve our test suites. Mutation realizes this idea and measures the confidence inspired by our testing. This method has reached a maturity phase and gradually gains popularity both in academia and in industry [1]. Figs. 1 and 2 record the current trends on the number of publications according to the data we collected (will be presented later). As demonstrated by these figures, the number of scientific publications relying on mutation analysis is continuously increasing, demonstrated in Fig. 1, and numerous of these contributions appear in the major software engineering venues, shown in Fig. 2. Therefore, mutation testing can be considered as a mainstream line of research.

The underlying idea of mutation is to force developers to design tests that explore system behaviors that reveal the introduced defects. The diverge types of defects that one can use, leads to test cases with different properties

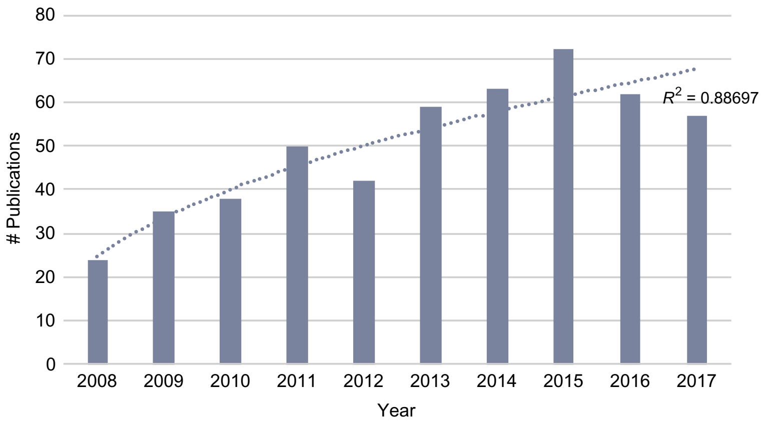


Fig. 1 Number of mutation testing publications per year (years: 2008–2017).

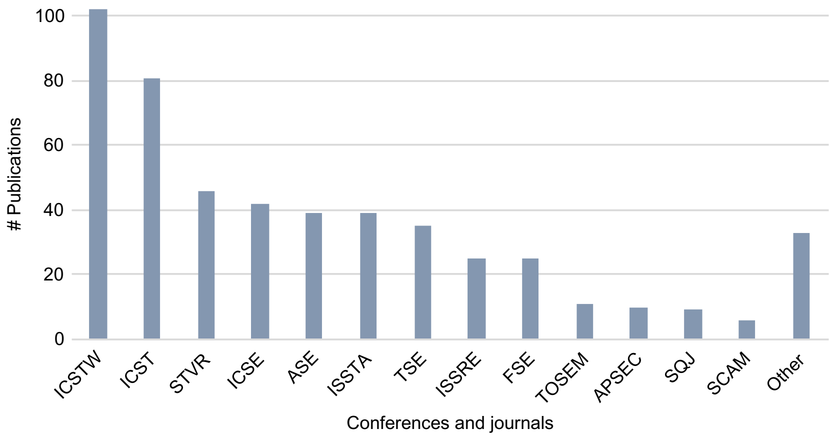


Fig. 2 Number of mutation testing publications per scientific venue.

as these are designed to reflect common programming mistakes, internal boundary conditions, hazardous programming constructs, and emulate test objectives of other structural test criteria. Generally, the method is flexible enough that it can be adapted to check almost everything that developers want to check, i.e., by formulating appropriate defects.

The flexibility of mutation testing is one of the key characteristics that makes it popular and generally applicable. Thus, mutation has been used for almost all forms of testing. Its primary application level is unit testing but several advances have been made in order to support other levels, i.e., specification [2], design [3], integration [4], and system levels [5]. The method has been applied on the most popular programming languages such as C [6], C++ [7], C# [8], Java [9], JavaScript [10], Ruby [11], including specification [2] and modeling languages [12]. It has also been adapted for the most popular programming paradigms such as object-oriented [13], functional [14], aspect-oriented, and declarative-oriented [15,16] programming.

In the majority of the research projects, mutation was used as an indicator of test effectiveness. However, recently researchers have also focused on a different aspect: the exploration of the mutant behaviors. Thus, instead of exploring the behavior space of the program under analysis the interest shifts to the behavior of the mutants. In this line of research, mutants are used to identify important behavior differences that can be either functional or nonfunctional. By realizing this idea, one can form methods that assist activities outside software testing. Examples of this line of research are methods that automatically localize faults [17], automatically repair software [18], and automatically improve programs' nonfunctional properties such as security [19], memory consumption [20], and execution speed [20,21].

Mutation analysis originated in early work in the 1970s [22–24], but has a long and continuous history of development improvement, with particularly important advances in breakthroughs in the last decade, which constitute the focus of this survey. Previous surveys can be traced back to the work of DeMillo [25] (in 1989), which summarized the early research achievements. Other surveys are due to Offutt and Untch [26] (in 2000) and Jia and Harman [27] (in 2011). Offutt and Untch summarized the history of the technique and listed the main problems and solutions at that time. Subsequently, Jia and Harman comprehensively surveyed research achievements up to the year 2009.

There is also a number of specialized surveys on specific problems of mutation testing. These are a survey on the equivalent mutant problem by Madeyski et al. [28] (in 2014), a systematic mapping of mutation-based test generation by Souza et al. [29] (in 2014), a survey on model-based mutation testing by Belli et al. [30] (in 2016), and a systematic review on search-based mutation testing by Silva et al. [31] (in 2017). However, none of these covers the whole spectrum of advances from 2009. During these years there are many new developments, applications, techniques, and advances in mutation testing theory and practice as witnessed by the number of papers we analyze (more than 400 papers). These form the focus of the present chapter.

Mutation testing is also increasingly used as a foundational experimental methodology in comparing testing techniques (whether or not these techniques are directly relevant to mutation testing itself). The past decade has also witnessed an increasing focus on the methodological soundness and threats to validity that accrue from such use of mutation testing and the experimental methodology for wider empirical studies of software testing. Therefore, the present chapter also collects together advices on best practices, giving the reader a "mini-handbook"-style roadmap for the application of mutation testing as an experimental methodology (in Section 9).

The present chapter surveys the advances related to mutation testing, i.e., using mutation analysis to detect faults. Thus, its focus is the techniques and studies that are related to mutation-guided test process. The goal is to provide a concise and easy to understand view of the advances that have been realized and how they can be used. To achieve this, we categorize and present the surveyed advances according to the stages of the mutation testing process that they apply to. In other words, we use the mutation testing process steps as a map for detailing the related advances. We believe that such an attempt will help readers, especially those new to mutation testing, understand everything they need in order to build modern mutation testing tools, understand the main challenges in the area, and perform effective testing.

The survey was performed by collecting and analyzing papers published in the last 10 years (2008–2017) in leading software engineering venues. This affords our survey a 2-year overlap in the period covered with the previous survey of Jia and Harman [27]. We adopted this approach to ensure that there is no chance that the paper could "slip between the cracks" of the two surveys. Publication dating practices can differ between publishers, and there is a potential time lag between official publication date and the appearance of a paper, further compounded by the blurred distinction between online availability and formal publication date. Allowing this overlap lends our survey a coherent, decade-wide, time window, and also aims to ensure that mutation testing advances do not go uncovered due to such publication date uncertainties.

Thus, we selected papers published in the ICSE, SigSoft FSE, ASE, ISSTA, ICST, ICST Workshops (ICSTW), ISSRE, SCAM, and APSEC conferences. We also collected papers published in the TOSEM, TSE, STVR, and SQJ journals and formed our set of surveyed papers. We augmented this set with additional papers based on our knowledge. Overall, we selected a set of 502 papers, which fall within five generic categories, those that deal with the code-based mutation testing problems (186 papers), those concerning model-based mutation testing (40 papers), those that tackle problems unrelated to mutation testing problems (25 papers), those that describe mutation testing tools (34 papers), and those that use mutation testing only to perform test assessment (217 papers). In an attempt to provide a complete view of the fundamental advances in the area we also refer to some of the publications that were surveyed by the two previous surveys on the topic, i.e., the surveys of Offutt and Untch [26] and Jia and Harman [27], which have not been obviated by the recent research.

The rest of the chapter is organized as follows: Section 2 presents the main concepts that are used across the chapter. Sections 3 and 4, respectively, motivate the use of mutation testing and discuss its relation with real faults. Next, the regular and other code-based advances are detailed in Sections 5 and 6. Applications of mutation testing to other artifacts than code and a short description of mutation testing tools are presented in Sections 7 and 8. Sections 9 and 10 present issues and best practices for using mutation testing in experimental studies. Finally, Section 11 concludes the chapter and outlines future research directions.

2. BACKGROUND

Mutation analysis refers to the process of automatically mutating the program syntax with the aim of producing semantic program variants, i.e., generating artificial defects. These programs with defects (variants) are called *mutants*. Some mutants are syntactically illegal, e.g., cannot compile, named "*stillborn*" mutants, and have to be removed. Mutation testing refers to the process of using mutation analysis to support testing by quantifying the test suite strengths. In the testing context, mutants form the objectives of the test process. Thus, test cases that are capable of distinguishing the behaviors of the mutant programs from those of the original program fulfill the test objectives. When a test distinguishes the behavior of a mutant (from that of the original program) we say that the mutant is "*killed*" or "*detected*"; in a different case, we say that the mutant is "*live*."

Depending on what we define as program behavior we can have different *mutant-killing conditions*. Typically, what we monitor are all the observable program outputs against each running test: everything that the program prints to the standard/error outputs or is asserted by the program assertions. A mutant is said to be killed *weakly* [32], if the program state immediately after the execution of the mutant differs from the one that corresponds to the original program. We can also place the program state comparison at a later point after the execution of a mutant. This variation is called *firm mutation* [32]. A mutant is *strongly killed* if the original program and the mutant exhibit some observable difference in their outputs. Thus, we have variants of mutation, called weak, firm, and strong. Overall, for weak/firm mutation, the condition of killing a mutant is that the program state has to be changed, while the changed state does not necessarily need to propagate to the output (as required by strong mutation). Therefore, weak mutation is expected to be less effective than firm mutation, which is in turn less effective than strong mutation. However, due to failed error propagation (subsequent computations may mask the state differences introduce by the mutants) there is no formal subsumption relation between any of the variants [32].

Mutants are generated by altering the syntax of the program. Thus, we have syntactic transformation rules, called "*mutant operators*," that define how to introduce syntactic changes to the program. For instance, an arithmetic mutant operator alters the arithmetic programming language operator, changing $+$ to $-$, for example. A basic set of mutant operators, which is usually considered as a minimum standard for mutation testing [33] is the five-operator set proposed by Offutt et al. [34]. This set includes the relational (denoted as ROR), logical (denoted as LCR), arithmetic (denoted as AOR), absolute (denoted as ABS), and unary insertion (denoted as UOI) operators. Table 1 summarizes these operators.

Defining mutant operators is somehow easy, we only need to define some syntactic alterations. However, defining useful operators is generally challenging. Previous research followed the path of defining a large set (almost exhaustive) of mutant operators based on the grammar of the language. Then, based on empirical studies, researchers tend to select subsets of them in order to improve the applicability and scalability of the method. Of course both the definition of operators and mutant selection (selection of

Table 1 The Popular Five-Operator Set (Proposed by Offutt et al. [34])

Names	**Description**	**Specific Mutation Operator**
ABS	Absolute value insertion	$\{(e, 0), (e, \texttt{abs}\ (e)), (e, \texttt{-abs}\ (e))\}$
AOR	Arithmetic operator replacement	$\{((\mathrm{a}\ op\ \mathrm{b}), a), ((\mathrm{a}\ op\ \mathrm{b}), \mathrm{b}), (x, y) \vert x, y \in \{+,-,*,/,\%\} \land x \neq y\}$
LCR	Logical connector replacement	$\{((\mathrm{a}\ op\ \mathrm{b}), a), ((\mathrm{a}\ op\ \mathrm{b}), b), ((\mathrm{a}\ op\ \mathrm{b}), \texttt{false}), ((\mathrm{a}\ op\ \mathrm{b}), \texttt{true}), (x, y) \vert x, y \in \{\&, \vert, \hat{}, \&\&, \Vert\} \land x \neq y\}$
ROR	Relational operator replacement	$\{((\mathrm{a}\ op\ \mathrm{b}), \texttt{false}), ((\mathrm{a}\ op\ \mathrm{b}), \texttt{true}), (x, y) \vert x, y \in \{>,>=,<,<=,==,!=\} \land x \neq y\}$
UOI	Unary operator insertion	$\{(cond, !\ cond), (v, -v), (v, \sim v), (v, --v), (v, v--), (v, ++v), (v, v++)\}$

representative subsets) form the two sides of the same coin. Since all possible operators are enormous if not infinite, the definition of small sets of them can be viewed as a subset selection (among all possible definitions). Here, we refer to mutant reduction as the process of selecting subsets of operators over a given sets of them. We discuss studies that define mutant operators in Section 5.1.1 and mutant reduction in Section 5.1.2.

Based on the chosen set of mutant operators, we generate a set of mutant instances that we use to perform our analysis. Thus, our test objectives are to kill the mutants (design test cases that kill all the mutants). We can define as "*mutation score*" or *mutation coverage* the ratio of mutants that are killed by our test cases. In essence, the mutation score denotes the degree of achievement of our test cases in fulfilling the test objectives. Thus, mutation score can be used as an adequacy metric [32].

Adequacy criteria are defined as predicates defining the objectives of testing [32]. Goodenough and Gerhart [35] argue that criteria capture what properties of a program must be exercised to constitute a thorough test, i.e., one whose successful execution implies no errors in a tested program. Therefore, the use of mutation testing as a test criterion has the following three advantages [6]: to point out the elements that should be exercised when designing tests, to provide criteria for terminating testing (when coverage is attained), and to quantify test suite thoroughness (establish confidence).

In practice using mutation score as adequacy measure, implicitly assumes that all mutants are of equal value. Unfortunately, this is not true [36]. In practice, some mutants are *equivalent*, i.e., they form functionally equivalent versions of the original program, while some others are *redundant*, i.e., they are not contributing to the test process as they are killed whenever other mutants are killed. Redundant mutants are of various forms. We have the *duplicated mutants*, mutants that are equivalent between them but not with the original program [37]. We also have the *subsumed mutants* [38] (also called *joint mutants* [39]), i.e., mutants that are jointly killed when other mutants are killed [36,39].

The problem with subsumed mutants is that they do not contribute to the test assessment process because they are killed when other mutants are also killed. This means that eliminating these mutants does not affect the selection/generation of test cases but the computation of the mutation score. Thus, the metric is inflated and becomes hard to interpret. As the distribution of mutants tend to be unpredictable and the identification of mutant equivalences and redundancies is an undecidable problem [37], it is hard to judge the test suite strengths based on the mutation score. In other words, the accuracy of the score metric is questionable. We will discuss this issue in depth in Section 9.

3. WHAT IS SPECIAL ABOUT MUTATION TESTING

Mutation testing principles and concepts share on a long heritage with more general scientific investigation, essentially drawing on the common sense of "trial and error" (that predates civilisation), and also resting on the foundations of inferential statistics and Popperian science [40].

One of the fundamental problems in software testing is the inability to know practically or theoretically when one has tested sufficiently. Practitioners often demand of researchers a method to determine when testing should cease. Unfortunately, this revolves around the question of what is intended by sufficiency; if we are to test in order to be sufficient to demonstrate the absence of bugs, then we are forced against the impossibility of exhaustive testing.

Faced with this challenge, much literature has centered on concepts of coverage, which measure the degree of test effort, that each coverage technique's advocates hope will be correlated with test achievement. For instance Table 2 reports on the main studies on this subject. There has been much empirical evidence concerning whether coverage is correlated with faults revelation [41–43], a problem that remains an important subject of study to the present day [6,41]. However, even setting aside the concern of whether such correlation exists, nonmutation-based forms of coverage suffer from a more foundational problem; they are essentially unfalsifiable (with respect to the goal of fault revelation), in the Popperian sense of science [40].

Mutation testing is important because it provides a mechanism by which assertions concerning test effectiveness become falsifiable; failure to detect certain kinds of mutants suggest failure to detect certain kinds of faults. Alternative coverage criteria can only be falsified in the sense of stating that should some desired coverage item remain uncovered, then claims to test effectiveness remain "false." Unfortunately, it is not usually possible to cover every desired coverage item, it is typically undecidable whether this criterion has been achieved in any case [59]. Coverage of all mutants is also undecidable [37], but mutation testing forms a direct link between faults and test achievements, allowing more scientific (in the sense intended by Popper) statements of test achievement than other less-fault-orientated approaches.

Mutation testing also draws on a rich seam of intellectual thought that is currently becoming more popular in other aspects of science and engineering. Such counterfactual reasoning can even be found beyond science, in the humanities, where historians use it to explore what would have happened had certain historical events failed to occur. This is a useful intellectual tool

Table 2 Summary of the Main Studies Concerned With the Relationship of Test Criteria and Faults

Author(s) [Reference]	Year	Test Criterion	Summary of Primary Scientific Findings
Frankl and Weiss [44,45]	1991, 1993	Branch, all-uses	All-uses relates with test effectiveness, while branch does not.
Offutt et al. [46]	1996	All-uses, mutation	Both all-uses and mutation are effective but mutation reveals more faults.
Frankl et al. [47]	1997	All-uses, mutation	Test effectiveness (for both all-uses and mutation) is increasing at higher coverage levels. Mutation performs better.
Frankl and Iakounenko [48]	1998	All-uses, branch	Test effectiveness increases rapidly at higher levels of coverage (for both all-uses and branch). Both criteria have similar test effectiveness.
Briand and Pfahl [49]	2000	Block, c-uses, p-uses, branch	There is no relation (independent of test suite size) between any of the four criteria and effectiveness.
Chen et al. [50]	2001	Block	Coverage can be used for predicting the software failures in operation.
Andrews et al. [42]	2006	Block, c-uses, p-uses, branch	Block, c-uses, p-uses, and branch coverage criteria correlate with test effectiveness.
Namin and Andrews [51]	2009	Block, c-uses, p-uses, branch	Both test suite size and coverage influence (independently) the test effectiveness.
Li et al. [52]	2009	Prime path, branch, all-uses, mutation	Mutation testing finds more faults than prime path, branch and all-uses.

Papadakis and Malevris [53]	2010	Mutant sampling, first- and second-order mutation	First-order mutation is more effective than second order and mutant sampling. There are significantly less equivalent second-order mutants than first-order ones.
Ciupa et al. [54]	2011	Random testing	Random testing is effective and has predictable performance.
Kakarla et al. [55]	2011	mutation	Mutation-based experiments are vulnerable to threats caused by the choice of mutant operators, test suite size, and programming language.
Wei et al. [56]	2012	Branch	Branch coverage has a weak correlates with test effectiveness.
Baker and Habli [57]	2013	Statement, branch, MC/DC, mutation, code review	Mutation testing helps improving the test suites of two safety-critical systems by identifying shortfalls where traditional structural criteria and manual review failed.
Hassan and Andrews [58]	2013	Multi-Point Stride, data-flow, branch	Def-uses is (strongly) correlated with test effectiveness and has almost the same prediction power as branch coverage. Multi-Point Stride provides better prediction of effectiveness than branch coverage.
Gligoric et al. [59,60]	2013, 2015	AIMP, DBB, branch, IMP, PCC, statement	There is a correlation between coverage and test effectiveness. Branch coverage is the best measure for predicting the quality of test suites.
Inozemtseva and Holmes [61]	2014	Statement, branch, modified condition	There is a correlation between coverage and test effectiveness when ignoring the influence of test suite size. This is low when test size is controlled.

Continued

Table 2 Summary of the Main Studies Concerned With the Relationship of Test Criteria and Faults—cont'd

Author(s) [Reference]	Year	Test Criterion	Summary of Primary Scientific Findings
Just et al. [43]	2014	Statement, mutation	Both mutation and statement coverage correlate with fault detection, with mutants having higher correlation.
Gopinath et al. [62]	2014	Statement, branch, block, path	There is a correlation between coverage and test effectiveness. Statement coverage predicts best the quality of test suites.
Ahmed et al. [63]	2016	Statement, mutation	There is a weak correlation between coverage and number of bug-fixes
Ramler et al. [64]	2017	Strong mutation	Mutation testing provides valuable guidance toward improving the test suites of a safety-critical industrial software system
Chekam et al. [6]	2017	Statement, branch, weak & strong mutation	There is a strong connection between coverage attainment and fault revelation for strong mutation but weak for statement, branch and weak mutation. Fault revelation improves significantly at higher coverage levels.
Papadakis et al. [41]	2018	Mutation	Mutation score and test suite size correlate with fault detection rates, but often the individual (and joint) correlations are weak. Test suites of very high mutation score levels enjoy significant benefits over those with lower score levels.

to help increase understanding and analysis of the importance of these events the influence (or forward dependence [65] as we might think of it within the more familiar software engineering setting. In software engineering, counterfactual reasoning plays a role in causal impact analysis [66], allowing software engineers to discover the impact of a particular event, thereby partly obviating the problem "correlation is not causation."

In mutation testing, we create a "counterfactual version of the program" (a mutant) that represents what the program would have looked like had it contained a specific chosen fault, thereby allowing us to investigate what would have happened if the test approach encounter a program containing such a fault. Causal impact analysis relies on recent development in statistics. In more traditional statistical analysis, mutation testing also finds a strong resonance with the foundations of sampling and inferential frequentist statistical analysis. A statistician might, for instance, seek to estimate the number of fish in a pool by catching a set of fish, marking these, and returning them to the pool, subsequently checking how many marked fish are present in a random sample. Of course such an approach measures not only the number of fish in the pool, but also the effectiveness of the recatching approach used in resampling. In a similar way, creating sets of mutant programs (marking fish) and then applying the test technique (resampling) can also be used to estimate the number of faults in the program, albeit confounded by concurrently measuring the effectiveness of the testing technique.

4. THE RELATIONS BETWEEN MUTANTS AND FAULT REVELATION

The underlying idea of mutation is simple; we form defects and we ask testers to design test cases that reveal them. Naturally, readers may ask why such an approach is effective. Literature answers this question in the following ways:

- First (theoretically), by revealing the formed defects we can demonstrate that these specific types of defects are not present in the program under analysis [67]. In such a case we assume that the formed mutants represent the fault types that we are interested in. In a broader perspective, the mutants used are the potential faults that testers target and thus, they are in a sense equivalent to real faults.
- Second (theoretically and practically), when test cases reveal simple defects such as mutants (defects that are the result of simple syntactic alterations), they are actually powerful enough to reveal more complex defects. In such a case, we assume that test cases that reveal the used types of defects also

reveal more complex types (multiple instances of the types we used) [68]. Thus, mutation helps revealing a broader class of faults (than those used) composed of the simple and complex types of faults that were used.

- Third (practically), when we design test cases that kill mutants we are actually writing powerful test cases. This is because we are checking whether the defects we are using can trigger a failure at every location (or related ones) we are applying them to. In such a case, we assume that test cases that are capable of revealing mutants are also capable of revealing other types of faults (different from the mutants). This is because mutants require checking whether test cases are capable of propagating corrupted program states to the observable program output (asserted by the test cases). Thus, potential faulty states (related to the mutants) have good chances to became observable [6].

The aforementioned points motivated researchers to study and set the foundation of mutation testing.

The first point has been shown theoretically based on the foundations of fault-based testing [67,69]. The practical importance of this assertion is that in case we form the employed mutants as common programming mistakes, then we can be confident that testers will find them. Thus, we can check against the most frequent faults. This premise becomes more important when we consider the competent programmer hypothesis [24], which states that developers produce programs that are nearly correct, i.e., they require a few syntactic changes to reach the correct version. This hypothesis implies that if we form mutants by making few simple syntactic changes we can represent the class of frequent faults (made by "competent programmers"). A recent study by Gopinath et al. [70] has shown that defects mined from repositories involve three to four tokens to be fixed, confirming to some extent the hypothesis. Generally, the recent studies have not consider this subject and thus, further details about the competent programmer hypothesis can be found in the surveys of Offutt and Untch [26] and Jia and Harman [27].

The second point is also known as the mutant coupling effect [68]. According to Offutt [68] the mutant coupling effect states "Complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will detect a large percentage of the complex mutants." Therefore, by designing test cases that reveal almost all the mutants used, we expect a much larger set of complex mutants to be also revealed. This premise has been studied both theoretically and practically (details can be found in the surveys of Offutt and Untch [26] and Jia and Harman [27]). Recent studies on the subject only investigated (empirically) the relationship between simple and complex mutants. For example the study of

Gopinath et al. [71] demonstrate that many higher order mutants are semantically different from the simple first-order ones they are composed of. However, the studies of Langdon et al. [72] and Papadakis and Malevris [53] show that test cases kill a much larger ratio of complex (higher order) mutants than simple ones (first-order ones) indicating that higher order mutants are of relatively lower strength than the first-order ones.

The third point is a realization of the requirement that the mutants must influence the observable output of the program (the test oracle). To understand the importance of this point we need to consider the so-called RIPR model (reachability, infection, propagation, revealability) [32]. The RIPR model states that in order to reveal a fault, test cases must: (a) reach the faulty locations (reachability), (b) cause a corruption (infection) to the program state (infection), (c) propagate the corruption to the program output (propagation), and (d) cause a failure, i.e., make the corrupted state observable to the user, be asserted by the test cases (revealability). Therefore, when designing test cases to kill mutants, we check the sensitivity of erroneous program states to be observable. Empirical evidence by Chekam et al. [6] has shown that this propagation requirement makes mutation strong and distinguishes it from weak mutation and other structural test criteria. In particular the same study demonstrates that mutant propagation is responsible for the revelation of 36% of the faults that can be captured by strong mutation.

Overall, the fundamental premise of mutation testing can be summarized as "if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault" [73]. This premise has been empirically investigated by the study of Chekam et al. [6] which demonstrated a strong connection between killing mutants and fault revelation. Similarly, the studies of Baker and Habli [57], Ramler et al. [64], and Ahmed et al. [74] have shown that mutation testing provides valuable guidance toward improving existing test suites.

A last reason that makes mutation strong is the fact that its test objectives are the formed defects. Thus, depending on the design of these defects, several test criteria can be emulated. Therefore, mutation testing has been found to be a strong criterion that subsumes, or probably subsumes[a] almost all other test criteria [32]. Thus, previous research has shown that strong mutation probably subsumes weak mutation [75], all data-flow criteria [47,55], logical criteria [76], and branch and statement criteria [52]. These studies suggest that a small set of mutant operators can often result in a set of test cases that is as strong as the ones resulting from other criteria.

[a] Subsumption is not guaranteed but it is probable to happen [32].

5. THE REGULAR CODE-BASED MUTATION TESTING PROCESS

This section details the code-based mutation testing advances. We categorize and present the surveyed advances according to the stages of the mutation testing process that they apply to. But before we begin the presentation we have to answer the following question: what is the mutation testing process? Fig. 3 presents a detailed view of the modern mutation testing process. This process forms an extension of the one proposed by Offutt and Untch [26]. The extension is based on the latest advances in the area. The important point here is that the process keeps the steps that are inherently manual outside the main loop of the testing process (steps in bold). The remaining steps can be sufficiently automated (although manual analysis may still be necessary).

Overall, the process goes as follows: first, we select a set of mutants that we want to apply (Step 1, detailed in Section 5.1) and instantiate them by forming actual executable programs (Step 2, detailed in Section 5.2). Next, we need to remove some problematic mutants, such as equivalent, i.e., mutants that are semantically equivalent to the original program despite being syntactically different, and redundant mutants, i.e., mutants that are semantically different to the original program but are subsumed by others (Step 3, detailed in Section 5.3).

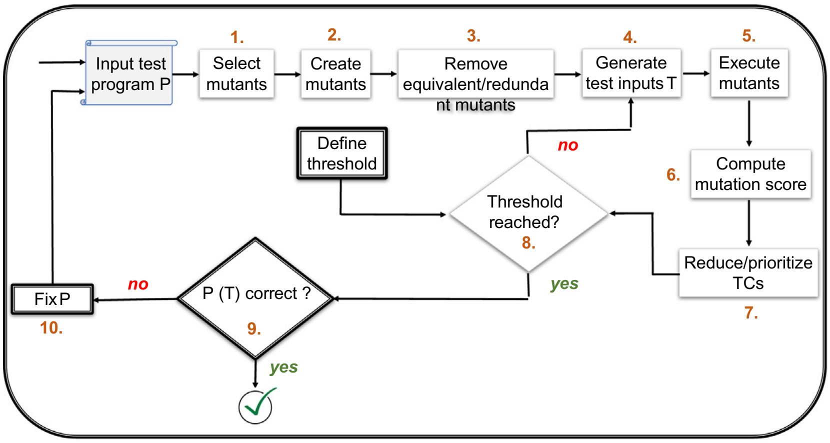


Fig. 3 Modern mutation testing process. The process forms an adaptation of the Offutt's and Untch's proposition [26] based on the latest advances in the area. *Bold boxes* represent steps where human intervention is mandatory.

Once we form our set of mutants we can generate our mutation-based test suite, execute it against our mutants and determine a score. In this step, we design (either manually or automatically) test cases that have the potential to kill our mutants (Step 4, detailed in Section 5.4) and execute them with all the mutants (Step 5, detailed in Section 5.5) to determine how well they scored (Step 6, detailed in Section 5.6). Subsequently, we perform test suite reduction by removing potentially ineffective test cases from our test suite. At this stage, we can also perform test case prioritization so that we order the most effective test cases first (Step 7, detailed in Section 5.7). The Steps 4–7 are repeated until the process results in a mutation score that is acceptable (Step 8, detailed in Section 5.8).

The last part of the process is when the user is asked to assess whether the results of the test executions were the expected ones (Step 9, detailed in Section 5.9). This step regards the so-called test oracle creation problem that involves the tester to assert the behavior of the test execution. In case faults are found then developers need to fix the problems and relaunch the process (Step 10, detailed in Section 5.10) until we reach an acceptable score level and cannot find any faults.

5.1 Mutant Selection (Step 1)

Mutation testing requires selecting a set of mutant operators based on which the whole process is applied. Thus, we need to define the specific syntactic transformations (mutant operators) that introduce mutants. Since defining mutant operators requires the analysis of the targeted language these may result in an enormous number of mutants. Therefore, in practice it might be important to select representative subsets of them. Section 5.1.1 refers to works that define mutant operators, while Section 5.1.2 refers to strategies that select subsets of mutants from a given set of mutants (aiming at reducing the cost of the process).

5.1.1 Mutant Operators

A large amount of work has focused on designing mutant operators that target different (categories of) programming languages, applications, types of defects, programming elements, and others.

5.1.1.1 Operators for Specific Programming Languages

Anbalagan and Xie [77] proposed a mutant generation framework for *AspectJ*, an aspect-oriented programming language. This framework uses two mutant operators; *pointcut strengthening* and *pointcut weakening*, which are used to increase or reduce the number of joint points that a pointcut matches.

Derezinska and Kowalski [8] introduced six object-oriented mutant operators designed for the intermediate code that is derived from compiled *C#* programs. Their work also revealed that mutants on the intermediate language level are more efficient than the high-level source code-level mutants.

Estero-Botaro et al. [78] defined 26 mutant operators for *WS-BPEL*—the Web Services Business Process Execution Language. Later on, they further quantitatively evaluated these operators regarding the number of stillborn and equivalent mutants each operator generates [79]. On the same topic, Boubeta-Puig et al. [80] conducted a quantitative comparison between the operators for *WS-BPEL* and those for other languages. The results indicate that many of the *WS-BPEL* operators are different due to the lack of common features with other languages (e.g., functions and arrays).

Mirshokraie et al. [10,81] proposed a set of *JavaScript* operators. These are designed to capture common mistakes in *JavaScript* (such as changing the *setTimeout* function, removing the *this* keyword, and replacing *undefined* with *null*). Experimental results indicate the efficiency of these operators in generating nonequivalent mutants.

Delgado-Pérez [7,82] conducted an evaluation of class-level mutant operators for *C++*. Based on the results, they propose a C++ mutation tool, MuCPP, which generates mutants by traversing the abstract syntax tree of each translation unit with the Clang API.

5.1.1.2 Operators for Specific Categories of Programming Languages

Derezinska and Kowalski [8] explored and designed the mutant operators for *object-oriented programs* through *C#* programs. They advocated that traditional mutant operators are not sufficient for revealing object-oriented flaws. Hu et al. [83] studied in depth the equivalent mutants generated by object-oriented class-level mutant operators and revealed differences between class-level and statement-level mutation: statement-level mutants are more easy to be killed by test cases.

Ferrari et al. [84] focused on *aspect-oriented programs*. They design a set of operators based on the aspect-oriented fault types. Similarly to the work of Anbalagan and Xie [77], this work uses *AspectJ* as a representative of aspect-oriented programs. Except for pointcut-related operators, operators for general declarations, advice definitions, and implementations are also adopted.

Bottaci [85] introduced an mutation analysis approach for dynamically typed languages based on the theory that the mutants generated by modifying types are very easily killed and should be avoided. Gligoric et al. [86] mentioned that "almost all mutation tools have been developed for statically typed languages," and thus proposed *SMutant*, a mutation tool that postpones mutation until execution and applies mutation testing dynamically instead of statically. In this way, the tool is able to capture type information of *dynamic languages* during execution.

5.1.1.3 Operators for Specific Categories of Applications

Alberto et al. [87] investigated the mutation testing approach for *formal models*. In particular, they introduced an approach to apply mutation testing to *Circus* specifications as well as an extensive study of mutant operators. Praphamontripong et al. [88,89] and Mirshokraie et al. [10] designed and studied mutant operators for web applications. Example operators are link/field/transition replacement and deletion. In a follow up study, Praphamontripong and Offutt [90] refined the initial set of operators (they exclude three operators), i.e., operators proposed in [89], by considering the redundancy among the mutants they introduce.

Deng et al. [91,92] defined mutant operators specific for the characteristics of *Android apps*, such as the event handle and the activity lifecycle mutant operators. Usaola et al. [93] introduced an abstract specification for defining and implementing operators for context-aware, mobile applications. Similarly, Linares-Vasquez et al. [94] introduced 38 mutation operators for Android apps. These operators were systematically derived by manually analysis types of Android faults. Oliveira et al. [95] proposed 18 *GUI-level* mutant operators. Also focused on GUI mutants, Lelli et al. [96] specially designed mutation operators based on the created fault model. Abraham and Erwig [97] proposed operators for *spreadsheets*. Dan and Hierons [98] introduced how to generate mutants aiming at floating-point comparison problems. Jagannath et al. [99] introduced how to generate mutants for actor systems.

Maezawa et al. [100] proposed a mutation-based method for validating Asynchronous JavaScript and XML (Ajax) applications. The approach is based on delay-inducing mutant operators that attempt to uncover potential delay-dependent faults. The experimental study suggests that by killing these mutants, actual errors can be revealed.

The study of Xie et al. [101] describes a mutation-based approach to analyze and improve parameterized unit tests (PUTs). The authors propose appropriate mutant operators that alter the effectiveness of the PUT test by varying the strength of its assumptions and assertions.

5.1.1.4 Operators for Specific Categories of Bugs

Brown et al. [102] proposed a technique to mine mutation operators from source code repositories. The intuition of this work is that by making mutants syntactically similar to real faults one can get semantically similar mutants.

Loise et al. [19] uses mutation testing to tackle security issues. They proposed 15 security-aware mutant operators for Java. Nanavati et al. [103,104] realized that few operators are able to simulate memory faults. They proposed nine memory mutant operators targeting common memory faults. Garvin and Cohen [105] focus on feature interaction faults. An exploratory study was conducted on the real faults from two open source projects and mutants are proposed to mimic interaction faults based on the study's results. Al-Hajjaji et al. [106] specially focus on variability-based faults, such as feature interaction faults, feature independency faults, and insufficient faults. Based on real variability-related faults, they derive a set of mutant operators for simulating them.

Additionally, other studies focus on the design of mutant operators for different levels or different programming elements. Mateo et al. [5] defined system-level mutant operators. Applying mutation at the system level faces two problems: first, mutating one part of the system can lead to an anomalous state in another part, thus comparing program behavior is not a trivial task; and, second, mutation's execution cost. To resolve these problems, the authors turn to weak mutation by introducing *flexible weak mutation* for the system level. The proposed approach is embedded in a mutation tool named Bacterio. Delamaro et al. [107] designed three new deleting mutant operators that delete variables, operators, and constants.

Additional categories of operators are due to Belli et al. [108], who proposed mutant operators for go-back functions (which cancel recent user actions or system operations), including basic mutant operators (i.e., transaction, state, and marking mutant operators), stack mutant operators (i.e., write replacement and read replacement), and high order mutant operators.

Gopinath and Walkingshaw [109] proposed operators targeting type annotations. This line of work aims at evaluating the appropriateness of type annotations. Jabbarvand and Malek [110] introduced an energy-aware framework for Android application. In this work, a set of 50 mutant operators mimicking energy defects was introduced. Arcaini et al. [111] proposed operators targeting regular expressions. These aim at assisting the generation of tests based on a fault model involving the potential mistakes one could made with regex.

5.1.2 Mutant Reduction Strategies

Mutant reduction strategies aim at selecting representative subsets from given sets of mutants. The practical reason for that is simply to reduce the application cost of mutation (since all the costly parts depend on the number of mutants).

Perhaps the simpler way of reducing the number of mutants is to randomly pick them. This approach can be surprisingly effective and achieve reasonably good trade-offs. Papadakis and Malevris [53] report that randomly selecting 10%, 20%, 30%, 40%, 50%, and 60% of the mutants results in a fault loss of approximately 26%, 16%, 13%, 10%, 7%, and 6%, respectively. Zhang et al. [112] reports that by killing randomly selected sets of mutants, composed of more than 50% of the initial set, results in killing more than 99% of all the mutants. Recently, Gopinath et al. [113] used large open source programs and found that a small constant number of randomly selected mutants is capable of providing statistically similar results to those obtained when using all mutants. Also, they found that this sample is independent of the program size and the similarity between mutants.

An alternative way of selecting mutants is based on their types. The underlying idea is that certain types of mutants may be more important than others and may result in more representative subsets than random sampling. Namin et al. [114] used a statistical analysis procedure to identify a small set of operators that sufficiently predicts the mutation score with high accuracy. Their results showed that it is possible to reduce the number of mutants by approximately 93%. This is potentially better than the mutant set of the previous studies, i.e., the five-operator set of Offutt et al. [34]. Investigating ways to discover relatively good trade-offs between cost and effectiveness, Delgado et al. [82] studied a selective approach that significantly reduce the number of mutants with a minimum loss of effectiveness for C++ programs. Delamaro et al. [107,115] experimented with mutants that involve deletion operations (delete statements or part of it) and found that they form a cost-effective alternative to other operators (and selective mutation strategies) as it was found that they produce significantly less equivalent mutants. In a later study, Durelli et al. [116] studied whether the manual analysis involved in the identification of deletion equivalent mutants differs from that of other mutants and found no significant differences. The same study also reports that relational operators require more analysis in order to asses their equivalence.

A later study of Yao et al. [117] analyzed the mutants produced by the five-operator set of Offutt et al. [34] and found that equivalent and stubborn mutants are highly unevenly distributed. Thus, they proposed dropping the ABS class and a subclass of the UOI operators (postincrement and decrement) to reduce the number of equivalent mutants and to improve the accuracy of the mutation score. Zhang et al. [118,119] conducted an empirical study regarding the scalability of selective mutation and found that it scales well for programs involving up to 16,000 lines of code. To further improve scalability, Zhang et al. [120] demonstrated that the use of random mutant selection with 5% of the mutants (among the selective mutant operators) is sufficient for predicting the mutation score (of the selective mutants) with high accuracy.

A comparison between random mutant selection and selective mutation was performed by Zhang et al. [112]. In this study, it was found that there are no significant differences between the two approaches. Later, Gopinath et al. [121] reached a similar conclusion by performing a theoretical and empirical analysis of the two approaches. The same study also concludes that the maximum possible improvement over random sampling is 13%.

Overall, as we discuss in Section 9, all these studies were based on the traditional mutation scores (using all mutants) that is vulnerable to the "subsumed mutant threat" [36]. This issue motivated the study of Kurtz et al. [122], which found that mutant reduction approaches (selective mutation and random sampling) perform poorly when evaluated against subsuming mutants.

Both random sampling and selective mutation are common strategies in the literature. Gligoric et al. [123] applied selective mutation to concurrent programs and Kaminski and Ammann [124] to logic expressions. Papadakis and Traon [17,125] adapt both of them for the context of fault localization and report that both random sampling and selective mutation that use more than 20% of all the mutants are capable of achieving almost the same results with the whole set of mutants.

Another line of research aiming at reducing the number of mutants is based on the notion of higher order mutants. In this case, mutants are composed by combining two or more mutants at the same time. Polo et al. [126] analyzed three strategies to combine mutants and found that they can achieve significant cost reductions without any effectiveness loss. Later, studies showed that relatively good trade-offs between cost and effectiveness can be achieved by forming higher order combination strategies [39,53,127]. In particular, Papadakis and Malevris [53] found that second-order strategies can achieve a reduction of 80%–90% of the equivalent mutants, with approximately 10% or less of test effectiveness loss. Similar results are reported in the studies of Kintis et al. [39], Madeyski et al. [28], and Mateo et al. [128], who found that second-order strategies are significantly more efficient than the first-order ones. Taking advantage of these benefits and ameliorate test effectiveness losses, Parsai et al. [129] built a prediction model that estimates the first-order mutation score given the achieved higher order mutation score.

Other attempts to perform mutant reduction are based on the mutants' location. Just et al. [130] used the location of the mutants on the program abstract syntax tree to model and predict the utility of mutants. Sun et al. [131] explored the program path space and selected mutants that are as diverse as possible with respect to the paths covering them. Gong et al. [132] selected mutants that structurally dominate the others (covering them results in covering all the others). This work aims at weak mutation and attempts to statically identify dominance relations between the mutants. Similarly, Iida and Takada [133] identify conditional expressions that describe the mutant-killing conditions, which are used for identifying some redundant mutants. Pattrick et al. [134] proposed an approach that identifies hard-to-kill mutants using symbolic execution. The underlying idea here is that mutants with little effect on the output are harder to kill. To determine the effect of the mutants on the program output, Pattrick et al. suggests calculating the range of values (on the numerical output expressions) that differ when mutants are killed. This method was latter refined by Pattrick et al. [135] by considering the semantics (in addition to numeric ones) of Boolean variables, strings, and composite objects.

Another attempt to reduce the number of mutants is to rank them according to their importance. After doing so, testers can analyze only the number of mutants they can handle based on the available time and budget by starting from the higher ranked ones. The idea is that this way, testers will customize their analysis using the most important mutants. In view of this, Sridharan et al. [136] used a Bayesian approach that prioritizes the selection of mutant operators that are more informative (based on the set of the already analyzed mutants). Along the same lines, Namin et al. [137] introduced MuRanker an approach that predicts the difficulty and complexity of the mutants. This prediction is based on a distance function that combines three elements; the differences that mutants introduce on the control-flow-graph representation (Hamming distance between the graphs), on the Jimple representation (Hamming distance between the Jimple codes), and on the code coverage differences produced by a given set of test cases (Hamming distance between the traces of the programs). All these together allow testers to prioritize toward the most difficult to kill mutants.

Mirshokraie et al. [10,81] used static and dynamic analysis to identify the program parts that are likely to either be faulty or to influence the program output. Execution traces are used in order to identify the functions that play an important role on the application behavior. Among those, the proposed approach then mutates selectively: (a) variables that have a significant impact on the function's outcome and (b) the branch statements that are complex. The variables with significant impact on the outcome of the functions are identified using the usage frequency and dynamic invariants (extracted from the execution traces), while complex statements are identified using cyclomatic complexity.

Anbalagan and Xie [77] proposed reducing mutants, in the context of pointcut testing of AspectJ programs, by measuring the lexical distance between the original and the mutated pointcuts (represented as strings). Their results showed that this approach is effective in producing pointcuts that are both of appropriate strength and similar to those of the original program.

Finally, mutant reduction based on historical data has also been attempted. Nam et al. [138] generated calibrated mutants, i.e., mutants that are similar to the past defects of a project (using the project's fix patterns), and compared them with randomly selected ones. Their results showed that randomly selected mutants perform similarly to the calibrated ones. Inozemtseva et al. [139] proposed reducing the number of mutants by mutating the code files that contained many faults in the past.

5.2 Mutant Creation (Step 2)

This stage involves the instantiation of the selected mutants as actual executables. The easiest way to implement this stage is to form a separate source file for each considered mutant. This approach imposes high cost as it requires approximately 3 s (on average) to compile a single mutant of a large project [37]. Therefore, researchers have suggested several techniques to tackle this problem.

The most commonly used technique realizes the idea of meta-mutation, also known as mutant schemata [140], which encodes all mutants in a single file. This is achieved by parameterizing the execution of the mutants [141]. The original proposition of mutant schemata involved the replacement of every pair of operands that participate in an operation with a call to a meta-function that functions as the operand [140]. The meta-functions are controlled through global parameters. This technique has been adopted by several tools and researchers, Papadakis and Malevris [141] use special meta-functions to monitor the mutant execution and control the mutant application. Wang et al. [142] and Tokumoto et al. [143] use meta-functions that fork new processes. Bardin et al. [144] and Marcozzi et al. [145,146] instrument the program with meta-functions that do not alter the program state, called labels, to record the result of mutant execution at the point of mutation and apply weak mutation.

Another approach involves bytecode manipulation [9,147]. Instead of compiling the mutants, these approaches aim at generating mutants by manipulating directly the bytecode. Coles et al. adopt such an approach for mutating Java bytecode [147]. Derezinska and Kowalski [8] and Hariri et al. [148] adopt the same approach for mutating the Common Intermediate Language of .NET and LLVM Bitcode, respectively.

Other approaches involve the use of interpreted systems, such as the ones used by symbolic evaluation engines [149]. A possible realization of this attempt is to harness the Java virtual machines in order to control and introduce the mutants [150]. Finally, Devroey et al. [151] suggested encoding all mutants as a product line. The mutants can then be introduced as features of the system under test.

5.3 Statically Eliminating Equivalent and Redundant Mutants (Step 3)

This step involves the identification of problematic mutants before their execution. This is a process that is typically performed statically. The idea is that some equivalent mutants, i.e., mutants that are semantically equivalent to the original program despite being syntactically different, and some redundant mutants, i.e., mutants that are semantically different to the original program but are subsumed by others, can be identified and removed prior to the costly test execution phase. By removing these "useless" types of mutants we gain two important benefits: first, we reduce the effort required to perform mutation and, second, we improve the accuracy of the mutation score measurement. Unfortunately, having too many "useless" mutants obscures the mutation testing score measurement by either overestimating or underestimating the level of coverage achieved. This last point is particularly important as it is linked to the decision of when to stop the testing process, i.e., Step 8 (Section 5.8).

5.3.1 Identifying Equivalent Mutants

Detecting equivalent mutants is a well-known undecidable problem [28]. This means that it is unrealistic to form an automated technique that will identify all the equivalent mutants. The best we can do is to form heuristics that can remove most of these mutants. One such effective heuristic relies on compiler optimization techniques [37,152]. The idea is that code optimizations transform the syntactically different versions (mutants) to the optimized version. Therefore, semantically equivalent mutants are transformed to the same optimized version. This approach is called Trivial Compiler Optimization

(TCE) and works by declaring equivalences only for the mutants that their compiled object code is identical to the compiled object code of the original program. Empirical results suggest TCE is surprisingly effective, being able to identify at least 30% of all the equivalent mutants.

Other techniques that aim at identifying equivalent mutants are of Kintis and Malevris [153–155] who observed that equivalent mutants have specific data-flow patterns which form data-flow anomalies. Thus, by using static data-flow analysis we can eliminate a large portion of equivalent mutants. This category of techniques includes the use of program verification techniques, such as value analysis and weakest precondition calculus. Program verification is used to detect mutants that are unreachable or mutants that cannot be infected [144,145].

A different attempt to solve the same problem is based on identifying killable mutants. This has been attempted using (static) symbolic execution [149,156]. Such attempts aim at executing mutants symbolically in order to identify whether these can be killable with symbolic input data. Other approaches leverage software clones to tackle this issue [157]. Since software clones behave similarly, their (non)equivalent mutants tend to be the same. Therefore, likely killable mutants can be identified by projecting the mutants of one clone to the other [157].

Literature includes additional techniques for the identification of equivalent mutants using dynamic analysis. These require test case execution and, thus, are detailed in the Step 6 (Section 5.6).

5.3.2 Identifying Redundant Mutants

Redundant mutants, i.e., mutants that are killed when other mutants are killed, inflate the mutation score with the unfortunate result of skewing the measurement. Thus, it is likely that testers will not be able to interpret the score well and end up wasting resources or performing testing of lower quality than intended [152]. To this end, several researchers have proposed ways to statically reduce redundancies.

Researchers have identified redundancies between the mutants produced by the mutant operators. The initial attempts can be found in the studies of Foster [158], Howden [159], and Tai [160,161] which claimed that every relational expression should only be tested to satisfy the $>$, $==$, and $<$ conditions. Tai [160,161] also argued that compound predicates involving n conditional AND/OR operators should be tested with $n + 2(2 * n + 3)$ conditions. Along the same lines, Papadakis and Malevris [149,162] suggested inferring the mutant infection conditions (using symbolic execution) of the mutants produced by all operators and simplify them in order to reduce the effort required to generate mutation-based test cases. This resulted in restricted versions for the Logical, Relational, and Unary operators.

More recently, Kaminski et al. [76,163] analyzed the fault hierarchy of the mutants produced by the relational operators and showed that only three instances are necessary. Just et al. [164] used a similar analysis and identified some redundancies between the mutants produced by the logical mutant operators. In a subsequent work, Just et al. [165] showed that the unary operator can be also improved. Putting all these three cases together, i.e., Logical, Relational, and Unary operators, results in significant gains in both required runtime execution (runtime reductions of approximately 20%) and mutation score accuracy (avoiding mutation score overestimation which can be as high as 10%). Along the same lines, Fernandes et al. [166] proposed 37 rules that can help avoiding the introduction of redundant mutants. Their results showed that these rules can reduce (on average) 13% of the total number of mutants.

All these approaches are based on a "local" form analysis, which is at the predicate level (designed for the weak mutation). Thus, applying them on strong mutation may not hold due to: (a) error propagation that might prohibit killing the selected mutants [141] and (b) multiple executions of the mutated statements caused by programming constructs such as loops and recursion [167,168]. Empirical evidence by Lindström and Márki [168] confirms the above problem and shows that there is a potential loss on the mutation score precision of 8%, at most.

Recently, Trivial Compiler Optimization (TCE) [37,152] has been suggested as a way to reduce the adverse effects of this problem. Similar to the equivalent mutant identification, TCE identifies duplicate mutant instances by comparing the compiled object code of the mutants. Empirical results have shown that TCE identifies (on average) 21% and 5.4% of C and Java mutants [152]. The benefits of using TCE is that it is conservative as all declared redundancies are guaranteed and deals with strong mutation.

Finally, Kurtz et al. [169] attempts to identify nonredundant mutants using (static) symbolic execution. The idea is that by performing differential symbolic execution between the mutants it is possible to identify such redundancies.

5.4 Mutation-Based Test Generation (Step 4)

According to the RIPR model [32], in order to kill a mutant we need test cases that reach the mutant, cause an infection on the program state, manifest the infection to the program output at an observable to the user point (asserted by the test cases). Formulating these conditions as requirements we can drive the test generation process. Currently, there are three main families of approaches aiming at tackling this problem named as (static) constraint-based test generation [149,170], search-based test generation [171,172], and concolic/dynamic symbolic execution [141,172]. Additional details regarding the automatic test generation and mutation-based test generation can be found in the surveys of Anand et al. [173] and Souza et al. [29].

5.4.1 Static Constraint-Based Test Generation

Constraint-based methods turn each one of the RIPR conditions into a constraint and build a constraint system that is passed to a constraint solver. Thus, the mutant-killing problem is converted to a constraint satisfaction problem [170]. Wotawa et al. [174] and Nica [175] proposed formulating the original and mutant programs (one pair at a time) as a constraint system and use solvers to search for a solution that makes the two programs differ by at least one output value. Kurtz et al. [169] adopted the same strategy in order to identify subsuming mutants.

Papadakis and Malevris [149,176] suggested formulating the RIPR conditions under selected paths in order to simplify the constraint formulation and resolution process. A usual problem of path selection is the infeasible path problem, i.e., paths that do not represent valid execution paths, which is heuristically alleviated using an efficient path selection strategy [149,176].

Other attempts are due to Papadakis and Malevris [177] and Holling et al. [156] who used out of the box symbolic execution engines (JPF-SE [178] and KLEE [179], respectively) to generate mutation-based test cases. These approaches instrument the original program with mutant-killing conditions that the symbolic execution engine is asked to cover (transforms the mutant-killing problem to code reachability problem). Riener et al. [180] suggested using bounded model-checking techniques to search for solutions (counter examples) that expose the studied mutants.

5.4.2 Concolic/Dynamic Symbolic Execution Test Generation

To overcome the potential weaknesses of the static methods, researchers proposed dynamic techniques such as concolic/dynamic symbolic execution. Similar to the static methods, the objective is to formulate the RIPR conditions. However, dynamic techniques approximate the symbolic constraints based on the actual program execution. Therefore, there is a need to embed the mutant-killing conditions within the executable program and guide test generation toward these conditions.

The first approach that uses concolic/dynamic symbolic execution is that of Papadakis et al. [177,181] that targets weak mutation. The main idea of this approach is to embed the mutant infection conditions within the schematic functions that are produced by the mutant schemata technique (described earlier in Section 5.2). This way, all the mutants are encoded into one executable program along with their killing conditions (mutant infection conditions). Subsequently, by using a concolic/dynamic symbolic execution tool we can directly produce test cases by targeting the mutant infection conditions. Similarly, Zhang et al. [182] and Bardin et al. [144,183] use annotations to embed the mutant infection conditions within the program under analysis. Along the same lines, Jamrozik et al. [184] augment the path conditions with additional constraints, similar to mutant infection conditions, to target mutants.

All the earlier approaches are actually performing some form of weak mutation as they produce test cases by targeting mutants' reachability and infection conditions. Performing weak mutation often results in tests that can strongly kill many mutants [32]. However, these tests often only kill (strongly) trivial mutants which usually fail to reveal faults [6]. To improve test generation, there is a need to formulate the mutant propagation condition on top of the reachability and infection conditions. This is complex as it involves the formulation of the two executions (the one of the original programs and the one of the mutants) along all possible execution paths. Therefore, researchers try to heuristically approximate this condition through search. Papadakis and Malevris [141] search the path space between the mutation point until the program output. This helps finding inputs that satisfy the propagation condition. Finally, another approach proposed by Harman et al. [172] searches the program input space using a constrained search engine (reachability and infection conditions are augmented with an extra conjunct to additional constraints).

5.4.3 Search-Based Test Generation

Other dynamic test generation techniques use search-based optimization algorithms to generate mutant-killing test cases. The idea realized by this class of methods is to formulate and search the program input domain under the guidance of a fitness function. The primary concern for these approaches is to define a fitness function that is capable of capturing the RIPR conditions and effectively identify test inputs that satisfy these conditions.

There are many different search-based optimization algorithms to choose from, but in the case of mutation, the most commonly used ones are the hill climbing [172,185] and genetic algorithms. As mentioned earlier, the main concern of these methods is the formulation of the fitness function. For instance, Ayari et al. [186] formulates the fitness as mutant reachability (distance from covering mutants) and Papadakis et al. [177,181] formulates the fitness as fulfillment of mutant infection conditions (distance from infecting mutants).

As with the already-presented techniques, formulating the propagation condition in the fitness function is not straightforward and thus, it is approximated by formulating indirect objectives. Fraser and Zeller [171,187] measure the mutants' impact (the number of statements with changed coverage, between a mutant and the original programs, along the test execution) to form the propagation condition. Papadakis and Malevris [188–190] measure the distance to reach specific program points which when impacted (covered by the original program execution and not by the mutant execution or vice versa) result in mutant killing. These are determined based on the mutants that have been killed by the past executions.

Patrick et al. [191] proposed a technique to evolve input parameter subdomains based on their effectiveness in killing mutants. The experimental evaluation of this approach suggests that it can find optimized subdomains whose test cases are capable of killing more mutants than test cases selected from random subdomains.

The most recent approaches try to formulate the mutant propagation condition by measuring the disagreement between the test traces of the original program and the mutants. Fraser and Arcuri [192] count the number of executed predicates that differ while Souza et al. [185] measure the differences in the branch distances between the test executions.

5.5 Mutant Execution (Step 5)

Perhaps the most expensive stage of mutation testing is the mutant execution step. This step involves the execution of test cases with the candidate test cases. Thus, given a program with n mutants and a test suite that contains m tests, we have to perform $n \times m$ program executions at maximum. For instance, consider a case where we have 100 mutants and a test suite that requires 10 s to execute for the original program. In this case, we expect that our analysis will complete in 1000 s. This makes the process time-consuming (since a large number of mutants is typically involved) and, thus, limits the scalability of the method.

Tests	Mutants			
	m_1	m_2	m_3	m_4
t_1	✓			✓
t_2	✓			✓
t_3			✓	✓
t_4			✓	
t_5			✓	✓

Fig. 4 Example mutant matrix.

To reduce this overhead, several optimizations have been proposed. We identify two main scenarios where the optimizations may appear. The first one, which we refer to as "Scenario A," regards the computation of mutation score, while the second one, which we refer to as "Scenario B," regards the computation of a mutant matrix (a matrix that involves the test execution results of all tests with all the mutants; an example appears in Fig. 4). This mutant matrix is used by many techniques such as the mutation-based fault localization [17], the oracle construction [171], test suite reduction [193], and prioritization [194].

The difference between the aforementioned scenarios is that when computing the mutation score (Scenario A) we only need to execute a mutant until it is killed. Therefore, we do not need to reexecute the mutants that have already been killed by a test case with other test cases. This simple approach achieves major execution savings. However, it does not apply on the second scenario where we need to execute all mutants with all test cases.

To illustrate the difference, consider the example mutant matrix of Fig. 4. To construct this mutant matrix (Scenario B), we need 20 executions (four mutants executed with the five tests). A naive approach for computing the mutation score (Scenario A) that does the same will also require 20 executions. However, mutation score calculation requires computing the number of mutants that are killed. Therefore, once a mutant is killed we do not need to reexecute it. In the above example, the tester will make four executions for t_1 (mutants m_1, m_2, m_3, and m_4), and he will determine that m_1 and m_4 are killed. Then, he will execute all the live mutants with t_2 (2 executions, mutants m_2 and m_3), and he will determine that none of them is killed. Then, he will execute the same mutants with t_3 (two executions, mutants m_2 and m_3) and will determine that m_3 is killed. For the last couple of test cases t_4 and t_5 he will execute only the mutant m_2. The sum of these executions is 10 which is greatly reduced compared to the initial requirement of 20 executions.

In the above analysis, we implicitly consider that there is an order of the test cases we are using. Therefore, by using different orders we can reduce further the number of test executions. In the example of Fig. 4, if we execute t_1 and t_3 first and then t_2, t_4, and t_5, we can reduce the number of test executions to 9. Zhang et al. [195] realized this idea using test case prioritization techniques. Similarly, Just et al. [196] proposed using the fastest test cases first and Zhu et al. [197] selected pairs of mutants and test cases to run together based on the similarity of mutants and test cases (identified by data-compression techniques). Of course, these two approaches only apply to Scenario A.

Regarding both Scenarios A and B, there are several optimizations that try to avoid executing mutants that have no chance of being killed by the candidate test cases. Thus, mutants that are not reachable by any test should not be executed as there is no chance of killing them. This is one of the initial test execution optimizations that has been adopted in the Proteum mutation testing tool [198]. This tool records the execution trace of the original program and executes only the mutants that are reachable by the employed tests. In practice, this optimization achieves major speed-ups compared to the execution of all tests (in both considered scenarios).

Papadakis and Malevris [177,181] observed that it is possible to record with one execution all the mutants that can be infected by a test. This was implemented by embedding the mutant infection conditions within the schematic functions that are produced by the mutant schemata technique (described in Section 5.2). Thus, instead of executing every mutant with every test, it is possible to execute all mutants at once, by monitoring the coverage of the infection conditions. Durelli et al. [150] suggested harnessing the Java virtual machine instead of schematic functions in order to record the mutant infection conditions. Both these methods resulted in major speed-ups (up to five times).

When performing strong mutation many mutants are not killed despite being covered by test cases, simply because the mutant execution did not infect the program state. Therefore, there is no reason to strongly execute mutants that are not reached and infected by the candidate test cases. Based on this observation, Papadakis and Malevris [141] proposed to strongly execute only the mutants that are reached and infect the program state at the mutant expression point. Along the same lines, Kim et al. [199] proposed optimizing test executions by avoiding redundant executions identified using statement-level weak mutation. Both the studies of Papadakis and Malevris [141] and Kim et al. [199] resulted in major execution savings. More recently, Just et al. [200] reported that these approaches reduce the mutant execution time by 40%. A further extension of this approach is to consider mutant infection at the mutant statement point (instead of mutant expression) [142].

Other test execution advances include heuristics related to the identification of infinite loops caused by mutants. Such infinite loops are frequent and greatly affect mutant execution time. However, since determining whether a test execution can terminate or not is an undecidable problem heuristic solutions are needed. The most frequent practice adopted by mutation testing tools to terminate test execution is with predefined execution time thresholds, e.g., if it exceeds three times the original program execution time. Mateo et al. [201,202] proposed recording program execution and determine whether potential infinite loops are encountered by measuring the number of encountered iterations.

All the aforementioned works aim at removing redundant executions. Another way to reduce the required effort is to take advantage of common execution parts (between the original program and the mutants). Thus, instead of executing every mutant from the input point to the mutation point, we can have a shared execution for these parts and then a different execution for the rest (from the mutation point to the program output). Such an approach is known as split-stream execution [203]. The separation of the execution is usually performed using a fork mechanism [143]. Empirical results suggest that such an approach can substantially improve the mutant execution process [142,143]. It is noted that these approaches are orthogonal to those that are based on mutant infection. Therefore, a combination of them can further improve the process and substantially enhance the scalability of the method, as demonstrated by Wang et al. [142].

An alternative way of speeding-up mutation testing is by leveraging parallel processing. This is an old idea that has not been investigated much. There are many tools supporting parallel execution of mutants, such as [147,204,205], but they do not report any results or specific advances. Mateo et al. [206] report results from five algorithms and shows that the mutant execution cost is reduced proportionally to the number of nodes that one is using.

Finally, there are also approaches tackling the problem in specialized cases. For instance, Gligoric et al. [207,208] suggest a method for the efficient state-space exploration of multithreaded programs. This work involves optimization techniques and heuristics that achieve substantial mutant execution savings. In the context of fault localization, Gong et al. [209] proposed a dynamic strategy that avoids executing mutants that do not contribute to the computation of mutant suspiciousness and achieves 32.4%–87% cost reductions. In the case of regression testing, Zhang et al. [210] identify the mutants that are affected by the program changes (made during regression) and executes only those in order to compute the mutation score. The affected mutants are identified with a form of slicing (dependencies between mutants and program changes). Wright et al. [211] uses mutant schemata and parallelization to optimize the test of relational database schemas. Zhou and Frankl [212] proposed a technique called *inferential checking* that determines whether mutants of database updating statements (INSERT, DELETE, and UPDATE) can be killed by observing the state change they induce.

5.6 Mutation Score Calculation and Refinement (Step 6)

The mutant execution aims at determining which mutants are killed and which are not. By calculating this number, we can compute the mutation score that represents the level of the test thoroughness achieved. Determining whether a test execution resulted in killing a mutant requires observing and comparing the program outputs. Thus, depending on what we define as a program output we can have different killing conditions. Usually, what constitutes the program output is determined by the level of granularity that the testing is applied to. Usually in unit testing the program output is defined as the observable (public access) return values, object states (or global variables), exceptions that were thrown (or segmentation faults), and program crashes. In system level, program output constitutes everything that the program prints to the standard/error outputs, such as messages printed on the monitor, behavior of user interfaces, messages sent to other systems, and data stored (in files, databases, etc.).

In the case of nondeterministic systems, it is necessary to define mutant-killing conditions based on a form of oracle that models the behavior of the obtained outputs. Patrick et al. [213] use pseudo-oracles to test stochastic software. Rutherford et al. [214] use discrete-event simulations (executable specifications) to define assertions and sanity checks that model how "reasonable" are the test execution results (distribution topology, communication failure, and timing) of distributed systems.

Observing and comparing the program outputs often requires a test driver that it is program specific and, thus, researchers usually approximate program outputs by observing a subset of it, usually defined by the test assertions (and program crashes). Alternative techniques involve the use of stubs, oracle data, log messages, and internal program states that will be detailed later on in Section 5.9. Mateo et al. [128] proposed *flexible weak mutation*, an approach for system-level mutation testing that considers mutants as killed when they result in corrupted object states. Object states are checked after the execution of every method call. Wu et al. [104] record execution paths and determine whether mutants cause any deviations from the original program's ones (execution of different paths).

Computing the mutation score requires the removal of equivalent mutants. As already discussed in Section 5.3, identifying equivalent mutants is a manual task that is partially addressed through static heuristics. Since the problem is important, there are some attempts to approximate the mutation score using dynamic heuristics. The idea is that mutants that are not killed by the tests but are capable of causing differences on the program state are likely to be killable [215]. This idea was initially introduced by Grun et al. [215] and, later, studied by the works of Schuler and colleagues [216–218]. Overall, these studies examined several heuristics that measure different types of impact (breaking program invariants, changed return values, altered control-flow and data-flow) and showed that measuring whether mutants cause deviations on the program execution forms the best option.

The use of mutants' impact provides opportunities to define mutant selection and classification strategies. Schwarz et al. [219] defined a mutant selection strategy by selecting a small set of mutants with high impact and diverge locations (all over the codebase). Mutant classification provides opportunities to achieve good trade-offs between effectiveness and efficiency. Papadakis and Traon [220,221] defined such strategies and found that mutant classification is beneficial when low-quality test suites are used.

Other attempts to refine and approximate the mutation score with the use of mutant classification are due to Kintis et al. [222,223]. Kintis et al. observed that killable mutants are likely to compose a higher order mutant that behaves differently than the first-order ones that it is composed of. Based on this observation, a mutant classification strategy that identifies 81% of the killable mutants with a precision of 71% was proposed.

5.7 Reduce/Prioritize Test Cases (Step 7)

This step involves the test suite reduction and/or test suite prioritization. Test reduction refers to the process of removing test cases that are somehow redundant, i.e., test cases that when removed from the test suites do not change the mutation score. Test prioritization refers to the process of ordering test cases in such a way that mutants are killed as early as possible.

Mutation-based test suite reduction has been suggested by Usaola et al. [224], using a greedy algorithm. The idea is to iteratively select the test cases that kill the maximum number of mutants that were not killed by the previously selected test cases. Hao et al. [225] used mutation to estimate a confidence level for the fault detection loss experienced due to the reduced test suites. Therefore, users can minimize their test suites using structural criteria and get an estimation of the potential fault detection capability loss based on the mutants.

Shi et al. [193] used mutants to reduce test suites and measured different trade-offs between reduced test suites and fault detection loss when reduced test suites kill fewer mutants than the original (nonreduced) ones. Similar to this work, Alipour et al. [226] proposed reducing (simplifying) individual tests rather than removing some of them and measured the trade-offs between reductions and fault detection loss.

Regarding test case prioritization, Lou et al. [194] studied two prioritization schemes; one based on the number of mutants killed and one based on the distribution of the killed mutants and found that prioritizing based on the number of killed mutants performs best. Nguyen et al. [227] proposed ordering first the test cases that kill the most mutants in order to support the audit testing of web service compositions. In their work, they considered only a subset of mutants, which is the ones that do not violate the explicit contract with the service under analysis.

5.8 Confidence Inspired by Mutation Score (Step 8)

The mutation testing process stops when mutation score reaches a user-specified threshold. In theory, this threshold reflects the level of confidence that developers have on the testing performed. Unfortunately, there are very few studies related to this subject, i.e., measuring the relationship between mutation score and fault revelation. Along these lines, Li et al. [52] experimented with mutation-adequate test suites (test suites that kill all killable mutants) and showed that these tests reveal more faults than the ones of structural testing criteria. This result is in line with the results of older studies that showed the superiority of mutation testing over other structural test criteria [27,55].

The most recent studies on the subject are those of Papadakis et al. [41] and Chekam et al. [6] that studied the fault revelation ability of mutation testing. The most important finding of the studies is that the “relationship between strong mutation and fault revelation exhibits a form of threshold behavior" [6] and that "achieving higher mutation scores improves significantly the fault detection." This means that there is a strong connection between mutation score and fault revelation only at higher mutation score levels (above a specific threshold). However, below that level, the mutation score is completely disconnected from fault revelation. In practice, this means that inadequate test suites that fail to reach relatively high mutation scores are vulnerable to noise effects and testers should not be confident on their testing (based on them). Perhaps more importantly, the same study shows that strong mutation-adequate test suites are capable of revealing at least 90% of the program faults [6].

The study of Tengeri et al. [228] suggests that mutation testing forms a good indicator of the expected number of defects in a system (number of real faults reported after the release of the system). Since these defects are those missed by the testing process they can be viewed as quality indicators of the test suite thoroughness.

Generally, it is important to consider the role of equivalent and redundant mutants when studying the relationship between mutation score and fault revelation. In practice, the existence of both equivalent and redundant mutants makes the evaluation of the exact mutation score value obscure, with the unfortunate effect of overestimating or underestimating the true score [36,152,229]. In particular, equivalent mutants tend to reduce the true mutation score, while redundant mutants have mixed effects. Therefore, reliably studying the mutant–fault relation requires, to some extend, adequate solutions for these problems.

Overall, we know very little regarding this fundamental aspect of software testing (confidence inspired by mutation score). Studies increasing our understanding on this respect are important and should form one of the main subjects addressed by future research. Similarly, studies addressing the equivalent and redundant mutant problems are also key to this problem.

5.9 Test Oracles (Step 9)

Once we create test inputs and reach the desire level of mutation score, we need to check whether the program under test behaves as expected. Additionally, we need to equip our tests with test oracles that assert the desired behavior for future use. This is the phase where we actually find faults (when the program does not behave as expected). Unfortunately, in the absence of formal specifications this task is carried out manually.

One of the first attempts to automate this process is due to Fraser and Zeller [171,187] who used mutants to guide oracle assertion creation. The idea is to check (and assert) the part of the program output that is responsible for killing the mutants. Fraser and Zeller [171,187] formulate this as a search problem and devised an automated approach that generates test assertions. Testers are then asked to validate these assertions. The same method has also been extended to identify relevant pre- and postconditions suitable for parameterized tests [101,230]. In the same vein, Knauth et al. [231] evaluated the quality of contracts (written in the Java Modeling Language) by mutating them.

The use of mutants in creating test oracles has been a common practice in automated test generation tools. Evosuite [232] adopts this practice for generating test oracles for Java programs. Yoshida et al. [233,234] use the same method to support test generation for C/C++ programs. Jahangirova et al. [235,236] use mutants to detect relevant observed state differences and abstract them into test oracles.

Mutants have also been used to drive the creation of oracle data (a set of variables that should be monitored during testing) [237,238]. In this work, internal program variables are monitored and ranked according to their ability to kill mutants. Similar to the work of Fraser and Zeller, mutants assist the creation and minimization of the oracle data. Jahangirova et al. [239] use test generation and mutation testing to assess and improve oracles (code assertions). Additional details regarding test oracles can be found in the survey of Barr et al. [240].

5.10 Debugging (Step 10)

Research on mutation-based debugging has followed two main directions, namely, fault localization and fault fixing. The former refers to the problem of locating the code areas that are responsible for a given failure while the later to the problem of automatically repairing the fault using the available test suite.

5.10.1 Mutation-Based Fault Localization

Mutation-based fault localization was introduce by Papadakis and Le Traon [17,241] with their work on the Metallaxis method. The underlying idea of Metallaxis is that mutants killed mostly by failing tests have a connection (interaction) with the program defects that caused the program failures. Thus, mutants killed mostly by failing tests provide indications regarding the faulty program locations. Empirical results on this approach demonstrated that mutation-based fault localization is significantly superior to other types of fault localization techniques, such as spectrum-based fault localization [17,242].

Metallaxis was later extended to support mutant reduction techniques, such as selective mutation [125] and has been released as an automated tool called Proteum/FL [204]. The idea of Metallaxis was later extended by Moon et al. [243], who introduce the MUSE method. MUSE works by checking whether mutants turn the failing test cases into passing or not. The difference from Metallaxis is that MUSE does not consider the mutants that are killed (have different outputs from the original program) by failing test cases but still they are not passing.

Other mutation-based fault localization techniques are those of Zhang et al. [244] who studied fault localization on the context of evolving programs and localized suspicious program edits. Hong et al. [245,246] extended MUSE for multilingual programs. Empirical results demonstrated that mutation-based techniques identify the faulty program locations (and edits) as the most suspicious statements. Another work of this type is that of Murtaza et al. [247,248]. In this work, it was observed that the test execution traces produced by mutants and faults are similar. Musco et al. [249] used mutants to approximate a causal graph. This approach realizes the idea of tracking causality in call graphs by exploring the test paths that lead to killing mutants.

5.10.2 Mutation-Based Fault Repair and Other Debugging Activities

Mutation has been used to support program repair activities by Debroy and Wong [250,251]. These works observe that many faults are fixed by simple syntactic transformations. Therefore, since mutants are simple syntactic transformations, they form potential patch candidates. The advantage of this technique is that it is simple and can be completely automated by a mutation testing tool.

Generally, automated fault repair is a large field of research with many specialized applications. Most of the approaches use genetic programming or constraint-based techniques to select and check whether special types of mutants can fix the underlying faults.

One of the first attempts in the area is due to Weimer et al. [18,252], who used genetic programming with statement deletion, statement insertion, statement replacement, and crossover mutant operators in order to support the automated bug fixing. Empirical results demonstrated that this approach can be particularly effective [253]. Later, Weimer et al. [254] leveraged mutation testing advances in order to improve the performance of fault fixing. In this work, the duality between mutation testing and fault repair is detailed along with potential opportunities for cross fertilization between the approaches. Other fault repair approaches combining search and mutation testing are by Tan and Roychoudhury [255], who introduce a regression repair technique that searches for mutants (using eight types of mutant operators) that fix regression faults.

There are many automated fault repair techniques that use some form of syntactic transformations (can be viewed as specialized higher order mutants) in order perform program repair. For instance, Long and Rinard [256] and Kim et al. [257] use syntactic patterns to perform program repair. However, as these approaches fall outside the scope of the present paper, the interested reader is redirected to a specialized survey on this subject [258].

Other debugging techniques relying on a form of mutation testing is Angelic debugging [259]. The idea of angelic debugging is to perform a form of data state mutation in order to correct the program execution. The idea is to identify the set of values that can substitute the program state (at runtime) and results in a form of "correct" execution.

6. ALTERNATIVE CODE-BASED MUTATION TESTING ADVANCES

This section details code-based testing advances that do not conform to the mutation testing process, as depicted in Fig. 3. These advances include predictions of the mutation score, gamification of mutation testing, the use of search algorithms, and diversity-aware testing techniques.

Mutation testing requires performing the mutant execution step which is expensive even when using the test execution optimizations that were discussed in Section 5.5. To deal with this problem, many researchers have proposed the use of alternative proxies to measure the fault revealing potential of test cases. Gligoric et al. [59,60] found that branch coverage measurements are strongly correlated with mutation scores. Therefore, they argued that branch coverage could be used as an alternative to real faults and mutation faults. Later, Gopinath et al. [62] conducted a large empirical study and found that statement coverage scores correlate strongly with mutation scores. Unfortunately, there are also studies demonstrating that coverage does not correlate well with mutant detection [61].

Zhang et al. [260] built a classification model that predicts the mutants that are killed without executing them. The model relies on a number of features related to mutants, tests, and coverage measures and predicts the mutant execution results with a relatively good precision (with over 0.85 precision and recall).

Designing mutation-based tests is a tedious and potentially boring task that most developers try to avoid. As testing requires the involvement of developers, their motivation is crucial. In view of this, the studies of Rojas and Fraser [261] and Rojas et al. [262] suggested making these activities entertaining by gamefying mutation testing. The game includes two main roles: the attacker and the defender. The former aims at creating subtle non-equivalent mutants and the latter at creating test cases to kill these mutants. Overall, this approach helps educating and motivating developers. It can also help crowdsourcing complex tasks, such as test generation and adequacy evaluation [262].

Search-based mutation testing forms an alternative approach to traditional mutation testing. Instead of selecting mutants from a predefined set of operators, it uses meta-heuristic search techniques to evolve and optimize the generation of higher order mutants [263]. The idea here is to seek tailored mutants that fit to the particular goals of the testers. Thus, search-based techniques can be employed in order to search the space of all possible mutants for those that are subtle (mutants that are killed by only few test cases) [264], representative of all mutants [38], and realistic (both semantically and syntactically close to the original program) [72].

There is a number of approaches that utilize mutant optimization: Jia and Harman study ways to combine first-order mutants so that they produce subtle faults [38,264] for C programs. Harman et al. [265] report that by using search it is possible to improve test effectiveness (between 5.6% and 12%) while enjoying 15% improved efficiency (in Java programs). Langdon et al. [72] used grammar-based, biobjective, strongly typed genetic programming to form realistic mutants for C programs. Along the same lines, Omar et al. [15] experiment with Java and AspectJ mutants. Their results demonstrate that it is possible to generate subtle higher order mutants [266,267]. Omar et al. [267,268] experimented with different ways of combining first-order mutants in order to improve the efficiency of the approach. They found that a form of local search performs best. Wu et al. [269] use higher order mutation to genetically improve the nonfunctional properties of the program under test. Their approach yields time and memory performance improvements of approximately 18% and 20%, respectively.

Finally, Shin et al. [270,271] proposed using mutants as a test suite diversity measure and defined a mutation-based diversity test criterion. The idea of this approach is to construct test cases that can distinguish every mutant from every other mutant. Thus, instead of trying to distinguish the behavior of the mutants from that of the original program, they proposed to distinguish the behavior of every mutant from all the others.

7. ADVANCES BEYOND CODE-BASED MUTATION TESTING

This section presents approaches that use mutants to support software engineering activities other than code-based testing. We first outline those belonging to the model-based testing, and continue with those related to security testing and, finally, other applications and testing approaches.

7.1 Model-Based Testing

An important line of mutation-based research regards its application to model artifacts. Older approaches called it specification-based mutation but the newer ones refer to it as model-based mutation. Here, we briefly discuss these approaches. For a detailed description and discussion on this subject, we point the reader to the specialized survey of Belli et al. [30].

Henard et al. [272] propose the use of mutation to test software product lines. The variability of software product lines and configurable systems is compactly represented by feature models. Therefore, the study of Henard et al. introduces mutant operators that mutate feature models (and their constraints). The idea is to transform the feature model into an equivalent logic formula, which is mutated using logical operators (using a tool called MutaLog [273]). Subsequently, the effectiveness of the selected configurations (to detect conformance faults of feature models) is evaluated based on their ability to detect mutants. In a later study, the feature model mutants were used to assist the automatic repair of feature models [274]. This is achieved by iteratively mutating the model under analysis until it reaches the desirable state.

An advantage of using feature model mutants is that by targeting them, it is possible to generate (automatically) a small set of test configurations. Henard et al. [275] applied this idea using search-based optimization methods in order to minimize the number of selected configurations and maximize the number of killed mutants. Filho et al. [276,277] used a multiobjective optimization approach to achieve several trade-offs between the number of selected configurations and number of killed mutants, their diversity, etc.

Generating feature model mutants with the aim of selecting test configurations has also been attempted by Arcaini et al. [278]. The difference of this method from that of Henard et al. is that mutants are introduced directly on the feature model under test (instead of the logic formula). The study also reports results from a test configuration generation technique that attempts to kill these mutants. The same approach was then used to evaluate the conformance of feature models to a software product line [279]. This approach also attempts to automatically detect and remove conformance faults from the feature model, similar to Henard et al. [274]. The difference is that since mutants are applied directly on the feature model the resulting models are expected to be easy to understand.

The study of Trakhtenbrot [280] focuses of testing statechart-based models for reactive systems. This approach is concerned with specific semantics of statechart models that are not aligned with the model's implementation. These semantics are the "zero-time" abstraction and "maximal parallelism," which are the subjects of mutation. Considering the conformance relation of action systems, Aichernig and Jöbstl [281] proposed a technique for encoding the semantics (of action systems) as constrains to be incorporated in the test conformance relations. These relations form the mutant killable conditions. Similarly, Aichernig et al. [282,283] developed a mutation-based test generation technique for UML state machines.

Devroey et al. [12] introduced the notion of featured model-based mutation analysis, a flexible formalism based on featured transition systems, which enable the optimized generation, configuration, and execution of mutants. The main idea behind this approach is to represent the model mutants as products of a software product line [151]. Based on this idea, the authors demonstrate that the technique can speed-up mutant execution up to 1000 times when compared to other behavioral model mutation approaches. Similarly, Belli and Beyazit [284] propose a mutant generation technique that attempts to limit the introduction of equivalent and duplicated mutants. The same approach aims at optimizing the test case execution by avoiding the comparison of the mutants with the original models.

El-Fakih et al. [285] present a mutation-based test case generation technique for extended finite state machines (EFSMs) that evaluates whether the EFSM under test conforms to user defined faults. As part of the technique, the EFSM under test is mutated and test cases able to kill the generated mutants are generated. Another technique, proposed by Su et al. [286], utilizes mutated GUI models for test case generation of Android applications. This particular approach mutates an autogenerated stochastic GUI model of the application, represented as a FSM, in order to search for better models that will result in different, and potentially better, event sequences compared to the original model. Finally, Aichernig et al. [287] combine property-based testing with model-based mutation testing to generate efficiently test suites that target specific coverage criteria based on EFSMs.

As for code-based mutation, detecting equivalent mutants at the model level is a tricky problem. When considering behavioral models such as automata, this problem can be formulated as a language equivalence problem. Indeed, if two automata accept the same language, then their traces are the same and no test case can distinguish them. Language equivalence is P-SPACE complete but efficient algorithms exist. Devroey et al. [3] compared one of such algorithms with two sorts of simulations: one that is completely random and one that exploits syntactic differences between the models to direct trace generation to infected states. Biased simulations proved to be efficient for strong mutation cased on large models while the exact approach was more interesting for weak mutation.

Belli et al. [108] present a mutation-based technique to test "go-back" functions modeled by pushdown automata. This approach uses mutant operators that affect the transitions, state, and stack of pushdown automata. Aichernig and Lorber [288,289] propose a model-based mutation testing technique for timed automata that tackles the state-space explosion problem caused when unfolding timed automata. The method improves the test execution using the unfolded structure of the original specifications. Larsen et al. [290] build on this work and further improve its efficiency. Zhou et al. [291] present a specification-based mutation approach to test safety-critical systems. This method defines mutant operators for the input output symbolic transition system modeling language and introduce a test case generation technique to create test cases based on these mutants. Adra et al. [292] study the application of mutation to agent-based systems. This approach defines mutant operators to address the properties of this type of systems.

Stephan et al. [293,294] present a technique that compares model-clone detection techniques for Simulink models using mutation. This approach introduces a set of structural mutant operators designed to compare model-clone detectors. The design of the operators was based on the authors' observations of potential model edit operations in publicly available models. In an extension of this work, Stephan et al. [295] present a taxonomy of Simulink mutant operators that represent realistic edit scenarios when modeling.

Although this taxonomy was created with the comparison of model-clone detectors in mind, the authors suggest that it can represent Simulink model mutations in general. Later, Pill et al. [296] developed a mutation testing framework for Simulink models, named SIMULTATE.

Testing model transformations using mutation has been attempted by Khan and Hassine [297]. In this approach the authors introduce specific mutant operators for the Atlas Transformation Language. Later, Troya et al. [298] focus on the same subject and presented an extensive set of mutant operators that uses both first and higher order mutation transformations [299]. Another study that tests model transformations using mutation is due to Aranega et al. [300], who focuses on how to support the generation of mutation-adequate test cases for checking model transformations. In this case, test cases are test models. The intuition behind the approach is that building a test model that is able to kill alive mutants from scratch is difficult, thus, the approach attempts to provide guidance to select some of the already available test models (test cases) to modify to kill the alive mutants.

Bartel et al. [301] focus on testing dynamically adaptive systems. These systems are governed by adaptation policies that incorporate how and when the system will adapt. The approach focuses on testing whether these adaptation policies are correctly implemented using mutation. Thus, a set of mutant operators is defined using a meta-model that represent the policy formalisms. The approach also suggests a specialization of these mutant operators for the case of action-based adaptation policies.

Mutation testing has also been applied on NuSMV models. Arcaini et al. [302] create mutants of such models and checks whether the NuSMV model advisor (an automatic static model review tool) can statically detect these mutants. Mutant operators for UML domain models have been defined by Kaplan et al. [303]. This study aims at generating test cases based on information provided by the domain model, expressed as a UML class diagram with invariants, and the use case model of the application under test. Fraser and Wotawa [304] present another model-based mutation approach aiming at determining property violations of a model. The approach relies on the notion of property relevance which relates test cases to model properties, in an attempt to connect the failing of a test case with a violation of a property.

The application of mutation testing at system requirements that are expressed in a natural language has also been attempted. Trakhtenbrot [305] introduced a semantic mutation approach that introduces mutants related to the intended meaning of the requirements (requirements expressed by predifined patterns) by altering the pattern of the requirements. This enables the use of the mutants for test generation and test evaluation.

Mutation testing has also been applied on Alloy models. Sullivan et al. [306] performed mutation testing in declarative programming paradigm (Alloy language) to support test case generation and showed that it is robust at revealing real faults.

Finally, mutation has been applied on aspect-oriented programs by mutating state models. Xu et al. [307] propose two strategies for generating tests from such models. The first one leverages structural information from the state model to generate test cases, whereas the second one is based on counterexamples generated by model-checking (counterexamples that form illegal sequences of events in the original model). The study of Lindström et al. [308] introduces a mutation-based approach to test aspect-oriented models. The approach proposes a set of mutant operators targeting specific features of aspect-oriented modeling. Abstract tests created to kill the generated mutants evaluate the modeling of cross-cutting concerns and the weaving process, as well.

7.2 Security Testing

This section presents mutation-based approaches related to security. More precisely, applications on testing security policies [309–313], regression testing of security policies [314] and testing security protocols [315], are shortly described.

Testing security policies using mutation has been suggested by Mouelhi et al. [310]. In this study, Mouelhi et al. propose a meta-model that captures different rule-based security policy formalisms. This meta-model forms the subject for mutation which is performed using a set of (proposed) generic operators that can simulate faults in the various instantiations of the model. Along the same lines, Mouelhi et al. [312] present another mutation-based technique that automatically transforms functional test cases into security test cases (test the security policy). In this approach, mutation is used for two purposes: to identify the subset of the functional test cases that are impacted by the security policy and to relate this subset's functional test cases to specific security policy rules.

Dadeau et al. [315,316] propose a mutation-based test generation and evaluation technique that validates an implementation of a security protocol that is written in the High-Level Security Protocol Language. These approaches propose mutant operators that introduce leaks in the security protocols and creates abstract test cases for HLPSL models by targeting/killing mutants.

Other attempts to test security policies are due to Bertolino et al. [311] who propose a fault model for history-based security policies. This study aims at policies written in the PolPA language and proposes modification rules that attempt to simulate faults that can occur in the implementation of the policy decision point (PDP) and target only the static behavior of the PDP. Elrakaiby et al. [309] and Nguyen et al. [313] attempt to test the obligation policy enforcement and delegation policies. These goals are achieved by using mutant operators that introduce changes in key elements of the obligation policy management and delegation features.

Finally, Hwang et al. [314] investigate test selection techniques for regression testing of security policies. They proposed three techniques toward this goal, one of which is based on mutation analysis. This particular technique first uses mutation analysis to correlate policy rules and tests cases and, subsequently, it applies test selection. The test selection is performed by selecting test cases that are correlated with rules involved with syntactic changes between the original policy and its mutants.

7.3 Supporting Adaptive Random Testing, Boundary Value Analysis, and Combinatorial Interaction Testing

Mutation testing has been used to support or extend several not mutation-based testing methods. Thus, it has been used to support adaptive random testing, boundary value analysis, and combinatorial interaction testing.

In the context of combinatorial interaction testing, Papadakis et al. [317] proposed mutating the constraints between the program input parameters. Thus, instead of selecting input combinations that satisfy the input constraints only, the authors proposed selecting the combinations that make the mutated constraints invalid. The underlying idea of this approach is that the difference between the original and the mutant constraints define some form of "boundary" conditions that trigger faults. Empirical results with faulty applications demonstrated that mutants have a stronger correlation with faults than the input parameter combinations.

Zhang et al. [318] proposed a mutation-based extension to boundary value analysis. The approach mutates some predicates of a given path condition in order to define boundary values. Similarly to Papadakis et al. [317], these values are the solutions that satisfy the path condition and at the same time differentiate the original predicate from its mutants. The authors also propose a way to generate test cases that cover these boundary conditions based on constrained combinatorial interaction testing.

Patrick and Jia [319,320] proposed a technique, named Kernel Density, to support adaptive random testing. This technique guides the test selection process based on the killed mutants. Thus, tests killing new mutants are considered to be more distant than those killing the same ones. Empirical results show that test cases selected by this approach kill more mutants than the ones selected by adaptive random testing.

7.4 Other Mutation-Based Applications

This section describes approaches tackling general software engineering problems. These include program analysis, software verification, code clones, defect prediction, and regression testing.

Mutation analysis has been used to automatically detect loop invariants by mutating postcondition clauses [321]. In such a way, many invariant candidates are generated and invalid invariants are discarded based on appropriate counterexamples. The study of Galeotti et al. [321] describes several ways to mutate the post conditions, as well as, ways to eliminate some trivial cases. Similarly, Andrés et al. [322] propose an automated framework, named PASTE, that uses mutants to evaluate the fault revealing ability of system invariants (generated from specifications) in the context of passive testing of stochastic timed systems. The approach evaluates the strengths of the invariants (and prioritizes them) based on the number of killed mutants. Subsequent works detail (its mutation module, its mutant operators and the algorithms it incorporates) extend and evaluate the framework further [323,324].

Pankumhang et al. [325] propose a code instrumentation technique, named iterative instrumentation, for measuring code coverage when testing time-sensitive systems. The approach is based on weak mutation analysis and instruments the program by inserting exit statements at the instrumentation points considered.

Other applications of mutation testing include its use for software verification. Groce et al. [326] used mutants to make developers familiar with software verification. The idea is to focus on incorrect programs (mutants) in order to understand when and how the verification process fails (by observing failures to detect problems caused by mutants).

Using mutation analysis to create software clones forms another application of mutation. Roy and Cordy [294] introduce several mutant operators that model typical copy/paste activities of developers and create clones based on the application of these operators. The same study also uses these clones to evaluate different clone detection techniques. Along these lines, the work of Svajlenko et al. [327] presents mutant operators that create fork constructs in order to assist the study and analysis of code similarity.

More recently, Bower et al. [328] proposed using mutation to assist the prediction of software defects. This technique combines traditional source code metrics with a number of mutation analysis metrics to built defect classifiers. The mutation analysis metrics that were used are classified into static, e.g., the number of mutants a mutant operator generated, and dynamic ones, e.g., the number of mutants killed. The study concludes that mutation-based metrics significantly improve the performance of defect prediction and that the best results are obtained by using a combination of static and dynamic metrics.

In the context of regression testing, Zhang et al. [329] used special forms of mutants to improve the fault detection ability of regression test suites. The approach mutates both the old and the new versions of the program under test and executes them with the available test suites. The detected differences between the two versions are considered as problems.

Di Nardo et al. [330] present a mutation-based technique to automatically generate faulty input data within complex data structures from existing field data. The approach uses six generic mutant operators that mutate the field data and guide their selection (using a data model). Results from an industrial case study show that it performs better, in terms of code coverage, than the manual testing performed by domain experts.

As discussed earlier, the existence of equivalent mutants constitutes one of the major costs of mutation's application. However, many researchers have started viewing this as an advantage in certain cases. Arcaini et al. [331,332] seek to find opportunities to improve the quality of the artifact under consideration. For example, they suggest that equivalent mutants can be used for improving code readability and for refactoring purposes. It is suggested that benefits from equivalent mutants may arise on all software artifacts where mutation applies. For instance, in feature models it is possible to detect dead and false optional features and redundant constraints. A similar approach is that of Baudry et al. [333], who suggested that equivalent mutants can be seen as diverse program versions. Therefore, by generating equivalent program versions, one can produce multiple diverse program variants (which can support security purposes, such as moving target defense).

Lisper et al. [334] introduced the concept of targeted mutation, which aims at nonfunctional properties. The idea underlying this approach is to introduce mutants that are relevant to a targeted nonfunctional property and use them as guides for generating and augmenting test suites. These test suites can then be used for estimating the worst-case execution time.

Finally, another mutation-based testing technique refers to testing relational database schemas. Wright et al. [211] investigate ways to make the application of mutation to database schemas more efficient. To this end, the authors propose and evaluate four cost-reduction approaches that leverage mutant schemata and parallelization. The results of the empirical study conducted suggest that the mutation analysis time can be reduced by the approaches proposed but they also indicate that their performance can be influenced by the underlining database management system (DBMS).

8. TOOLS FOR MUTATION TESTING

One important factor for the successful application of mutation is the availability of automated frameworks that support its application steps. This section discusses the tools that were introduced or were used in the studies we surveyed. Table 3 outlines the corresponding tools along with the year of their creation, their application artifact, and a concise description of key characteristics.

As it can be seen from the table, our analysis concluded in 76 tools, most of which where introduced between 2008 and 2017, that apply mutation to different software artifacts. By closely examining the table, it becomes obvious that there is an increasing growth in mutation testing tools with the creation of approximately 10 tools per year. Most of these tools target the implementation level languages but there are also tools that target specification languages and models.

At the implementation level, the mutation testing tools target mostly the C and Java programming languages. Most of the tools focus on the support of traditional, method-level mutant operators, and strong mutation, with few tools supporting object-oriented operators and weak or higher order mutation. Additionally, there have been various tools proposed that apply mutation to dynamically typed programming languages and concurrency-related aspects.

For the noncode-based tools, there have been proposed various tools for many model notations, including Extended and Timed Finite State Machines, Simulink models, Feature Models, etc., that automate the application of mutation. Furthermore, automated frameworks for mutating security policies and protocols have also been introduced.

9. MUTATION-BASED TEST ASSESSMENT: USE AND THREATS TO VALIDITY

Mutation testing is a popular technique for assessing the fault revealing potential of test suites. Much work on empirical software engineering relies

Table 3 Mutation Testing Tools

Name and Ref	Year	Application	Description
`mutate` [335]	N/A	C	Supports method-level mutant operators
`Jester` [336]	2001	Java	Supports source-code-level (src-level) mutant generation
`Proteum` [337]	2001	C	Supports an extensive set of method-level mutant operators and *interface mutation* (intermethod-level operators)
`mutgen` [338,339]	2003	C	Supports method-level mutant operators
`muJava` [9,340,341]	2004	Java	Implements src-level mutant generation and supports method-level and object-oriented (OO) mutant operators
`ByteME` [342]	2006	Java	Implements bytecode-level mutant generation and supports method-level and object-oriented (OO) mutant operators
`SQLMutation` [343]	2006	SQL	Supports traditional and SQL-specific mutant operators for SQL queries
`Jumble` [344]	2007	Java	Implements bytecode-level mutant generation and supports method-level mutant operators
`ESTP` [345]	2008	C	Supports 20 traditional C mutant operators
Not named [346]	2008	Sulu	Supports method-level mutant operators (drawn from the study of Andrews et al. [339])
`Milu` [347]	2008	C	Supports method-level mutant operators and higher order mutation
Not named [304]	2008	NuSMV models	Supports specification-based mutation (drawn from the study of Black et al. [2])

Continued

Table 3 Mutation Testing Tools—cont'd

Name and Ref	Year	Application	Description
Not named [348]	2008	Code generated from Simulink models	Seeds faults into the implementations generated from Simulink models
Not named [310]	2008	Security policies	Supports security-policy-access-control meta-model mutation (applied on policies defined in various notations (e.g., RBAC and OrBAC)
Not named [349]	2008	LOTOS specifications	Supports mutation testing for LOTOS specifications
Not named [77]	2008	AspectJ	Supports mutant operators for the creation of *pointcut* mutants that vary the strength of the corresponding pointcut in terms of the number of joint points it matches
`Javalanche` [205]	2009	Java	Implements bytecode-level mutant generation and supports method-level mutant operators and mutant classification based on mutants' impact
`JDama` [350,351]	2009	SQL/JDBC	Implements bytecode-level mutant generation and supports SQL-related operators and weak mutation
`AjMutator` [352,353]	2009	AspectJ	Supports mutant operators for AspectJ Pointcut Descriptors (PCDs) [84] and automated equivalent mutant detection
`GAmera` [354]	2009	WS-BPEL	Supports mutation testing for WS-BPEL composition
Not named [124]	2009	boolean logic	Supports mutant operators for possible DNF logic faults
`PASTE` [322–324]	2009	TFSM	Supports passive testing of systems presenting stochastic-time information using mutant operators specific to timed finite state machines (TFSM)

Not named [355]	2009	Z	Supports mutant operators for Z specifications
Not named [356]	2009	GCC-XML	Supports a subset of mutant operators proposed by Ellims et al. [357]
Not named [358]	2009	LUSTRE / SCADE	Supports mutant operators for LUSTRE/SCADE programs
Not named [231]	2009	Java	Supports mutant operators that follow the fault classification of Durães and Madeira [359]
`PIT` [147]	2010	Java	Implements bytecode-level mutant generation and supports method-level mutant operators
`MutMut` [208]	2010	Java	Supports concurrency-related mutant operators
`GenMutants` [182]	2010	.Net	Supports method-level mutant operators
`Judy` [360]	2010	Java	Implements src-level mutant generation and supports method-level and OO mutant operators
`webMuJava` [88]	2010	HTML/JSP	Supports specific mutant operators for web components written in HTML and JSP languages
`Bacterio` [5]	2010	Java	Supports method-level mutant operators for system-level testing using flexible weak mutation
Not named [141,177,181,188,190]	2010		Supports method-level mutant operators
`Major` [361]	2011	Java	Supports method-level mutant operators
`Para` μ [362]	2011	Java	Supports OO and concurrency-related mutant operators and higher order mutation

Continued

Table 3 Mutation Testing Tools—cont'd

Name and Ref	Year	Application	Description
ILMutator [8]	2011	C#	Implements CLI-level mutant generation and supports method-level and OO mutant operators
SMutant [86]	2011	Smalltalk	Supports traditional, method-level mutant operators in a dynamically typed language
MuBPEL [363]	2011	WS-BPEL	N/A
jMuHLPSL [315]	2011	HLPSL	Supports mutant operators that introduce leaks in security protocols
Not named [364]	2011	SPADE	Mutates the flow pattern description of input and output streams and the SPADE code of components
Not named [365]	2011	Aglets	Supports mutant operators specific to mobile agent systems that affect the movement, communication, run method, creation, event listeners, and agent proxy of an agent
Not named [366]	2011	Java	Supports method-level mutant operators based on the selective mutation approach and higher order mutation
SMT-C [367]	2012	C	Supports the semantic-related and method-level mutant operators
mutant (muRuby) [11,368]	2012	Ruby	Supports Ruby-specific mutant operators
Not named [309]	2012	Obligation policies	Supports mutant operators specific to obligation policy enforcement
Not named [149]	2012		Supports traditional, method-level mutant operators

CCMUTATOR [369]	2013	C/C++	Supports concurrency-related mutant operators, higher order mutation, and target applications written using the PThreads and C++11 concurrency constructs
Comutation [123]	2013	Java	Supports concurrency-related mutant operators [370]
SchemaAnalyst [371]	2013	SQL	Supports mutant operators related to relational schema integrity constraints, applied to multiple database management systems
XACMUT [372]	2013	XACML	Supports mutant operators targeting XACML 2.0 security policies
Mutandis [10,81]	2013	JavaScript	Supports JavaScript-specific mutant operators
Not named [373]	2013	Web service compositions	Supports two types of mutant operators for web service compositions: one that is internal to the service and one that models inconsistencies across different services of the composition
Not named [313]	2013	Security policies	Supports mutant operators specific to delegation policies based on a formal analysis of key delegation features
Not named [272]	2013	Feature models	Supports mutant operators for mutating feature models
MutPy[374]	2014	Python	Implements traditional and python-specific mutation operators
MuCheck [14]	2014	Haskell	Supports mutant operators targeting functional constructs and higher order mutation
HOMAJ [375]	2014	AspectJ/Java	Supports higher order mutation
Not named [376]	2014	HTML/CSS	Supports mutant operators that seed presentation defects to web pages

Continued

Table 3 Mutation Testing Tools—cont'd

Name and Ref	Year	Application	Description
Not named [377]	2014	EFSM	Supports mutants that introduce single transfer faults (STFs) and double transfer faults (DTFs) to extended finite state machines (EFSM) models
Not named [378]	2014	Data-flow languages	Supports two mutant operators that model common mistakes when creating power plant control programs
MutaLog [273]	2014	Logic mutation	Supports mutant operators for mutating logic expressions
`REDECHECK` [379]	2015	HTML/CSS	Supports mutant operators for layout defects in responsive web sites
Not named [380]	2015	Spreadsheets	Supports mutant operators for spreadsheets (*spreadsheet mutation*)
Not named [381]	2015	FSM	Supports mutant operators for FSM specifications (based on the studies of Fabbri et al. [382] and Petrenko et al. [383])
Not named [384]	2015	Component-level sequence and state diagrams	Supports architecture- and design-level mutant operators
Not named [385]	2015	HTML/JavaScript	Supports mutant operators for the Model-View-Controller frameworks of web application development
Not named [103,104]	2015	C	Supports memory-related mutant operators that model memory faults and control flow deviation as a mutant-killing condition
Not named [91]	2015	Android apps	Supports android-specific mutant operators, affecting intents, events, activity lifecycle, and XML files; and the method-level mutant operators of `muJava`
`MoMut` [386]	2015	UML models	Supports model-based mutation testing for UML state charts, class diagrams, and instance diagrams

MuVM [143]	2016	C	Implements bitcode-level mutant generation and supports higher order mutation
Not named [387]	2016	FBD	Supports mutant operators for FBD language
Not named [388]	2016	Simulink	Supports mutant operators that model common Simulink fault patterns
Not named [233]	2016	C++	Supports mutant operators similar to the ones of PIT for the Java language
Not named [148]	2016	C	Implements LLVM-level mutant generation and supports method-level mutant operators typically used in other tools, e.g., Milu
Not named [36,389]	2016	C	Supports traditional, method-level mutant operators
Not named [242]	2016		Supports traditional, method-level mutant operators
Vibes [12,390]	2016	Transition systems, Statechart models	Implements featured model-based mutation analysis and supports the Fabbri et al. [382] operator set (both first order and higher order)
*μ*Droid [110]	2017	Android apps	Implements energy-aware mutation operators derived from a specific energy defect model
MDroid+[94]	2017	Android apps	Implements mutation operators to test Android applications based on a specifically designed Android fault model derived from manual analysis of various software artifacts
Not named [102]	2017	Source code	Extracts mutation operators from changes made in the development history of projects in an attempt to produce more "realistic" mutants
LittleDarwin [391]	2017	Java	Supports method-level mutation operators, higher order mutation, mutant sampling, and disjoint/subsuming mutant analysis

Continued

Table 3 Mutation Testing Tools—cont'd

Name and Ref	Year	Application	Description
MuCPP [7]	2017	C++	Implements class-level, object-oriented mutation operators for C++ programs
MutRex [111]	2017	Regular expressions	Implements mutation operators based on a specific fault model for regular expressions
BacterioWeb [93]	2017	Android apps	Implements mutation operators for Android applications
Not named [6]	2017	C	Supports method-level mutant operators
Not named [142]	2017	C	Implements method-level mutation operators and the AccMut approach [142] to reduce the cost of mutant execution
Not named [19]	2017	Java	Implements security-aware mutation operators

on the use of artificial faults (mutants or manually seeded faults). Researchers employ mutants to perform controlled experiments and assess the relative strengths of test strategies. We call this practice as *mutation-based test assessment.*

A typical mutation-based test assessment scenario arises when we want to determine whether one method, say Method-1, is more effective than another one, say Method-2. For instance, suppose that the methods to compare are a random test generation (Method-1) and a search-based test generation (Method-2). In this case, our objective is to check whether one of them has a higher fault revealing ability within a given amount of time. This is assessed by counting the number of killed mutants, i.e., the technique that kills the highest number of mutants is the winning one. Of course, in this particular case, the approaches are stochastic and thus, the experiment needs to be repeated multiple times and assessed by a statistical test, but in every case the technique that kills statistically significant more mutants is the winning one.

This is an intuitive choice made by many empirical studies. However, is it safe to conclude that Method-1 which kills more mutants than Method-2 is better? Actually, it is hard to draw any such conclusion unless we carefully consider and control a number of parameters. As we shall discuss in this section, there are influential factors lying at the heart of mutation-based test assessment that can hamper our ability to assess the fault revealing potential of the techniques.

9.1 The Use of Mutation in Empirical Studies

Using mutants as an effectiveness metric is a common practice, which previous research suggests that is adopted by more than a quarter of software testing controlled experiments [36]. To demonstrate the importance and popularity of this practice, we collected the papers that conduct empirical studies and use mutation testing as an assessment method and we analyze them to identify: (1) how often mutation testing is used as an assessment method; (2) the types of assessment that are used; (3) the tools that are frequently used; and (4) the languages mutation is applied to.

The results of our analysis show that, in most cases, mutation was applied to evaluate test techniques. Thus, mutants are used as proxies for real faults and the mutation score is used as an indicator of real fault detection. Fig. 5 presents the distribution of these studies in a yearly basis, including the overall growth trend. It is clear from the figure that the application of mutation testing, to experimental studies has been steadily increasing over the last 10 years.

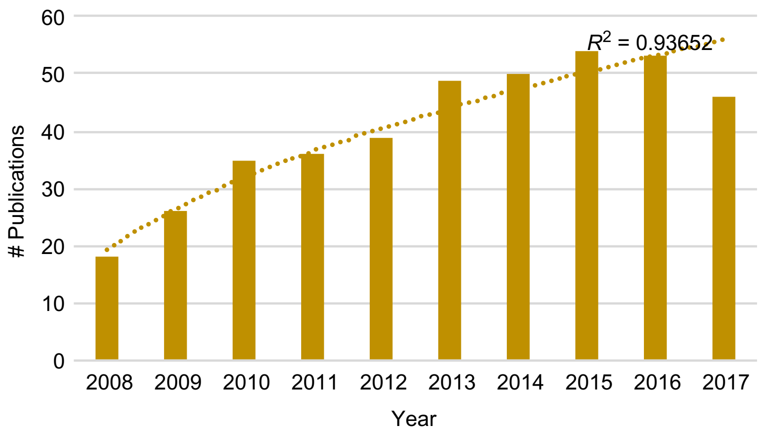


Fig. 5 Number of empirical studies using mutation testing as an assessment method.

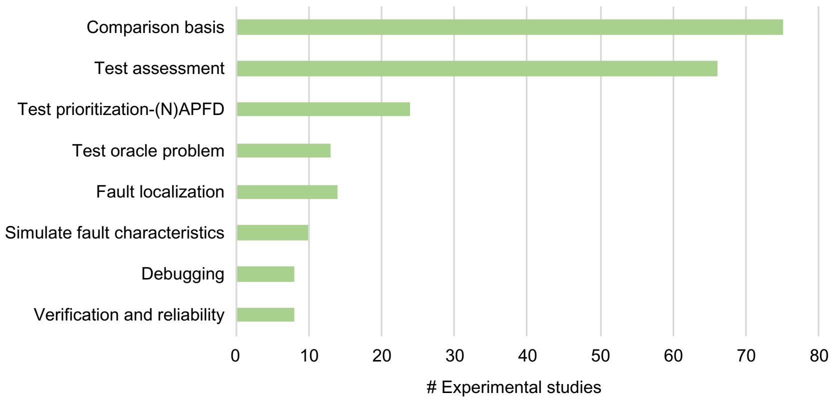


Fig. 6 Different types of mutation-based test assessment.

It is important to note that many of the analyzed studies are strictly not concerned with mutation testing; their objectives do not include mutation-related software engineering problems. Rather, mutation is a mechanism to validate the study and not the subject of the study.

Overall, from the papers we surveyed, we identified 190 papers falling into this category. Taking these findings into account, we can conclude that an increasing number of scientific results rely on mutation.

Fig. 6 presents the types of mutation-based test assessment. As can be seen from the figure, mutation's primary use in experimental studies is for comparing test techniques, *Comparison Basis*. The second largest category refers to *Test Assessment* and includes the test effectiveness of single techniques (without comparison), i.e., a test technique kills this number of

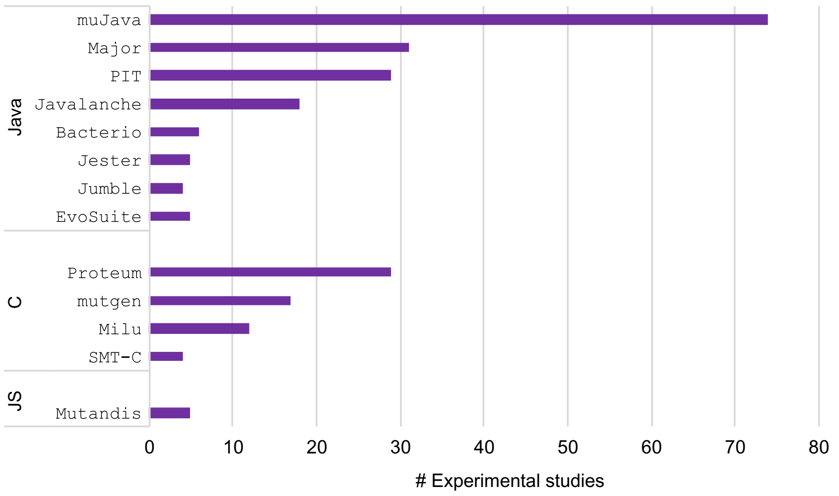


Fig. 7 Distribution of mutation testing tools in experimental studies.

mutants. The other categories involve assessments for test prioritization (*Test Prioritization-(N)APFD*),[b] test oracle (*Test Oracle Problem*), localization of faults (*Fault Localization*), verification techniques (*Verification and Reliability*), debugging (*Debugging*), and other techniques.

Fig. 7 depicts the mutation testing tools that were used in the experimental studies along with the number of studies using these tools. It is noted that the figure includes only the most frequently used tools.[c] As can be seen from the figure, `muJava` [9] and `Proteum` [198] are the most frequently used tools in experimental studies (operate on Java and C, respectively). Other frequently used Java mutation systems are `PIT` [147], `Major` [361], and `Javalanche` [205]. For C, the `mutgen` framework, used in the study of Andrews et al. [339] and `Milu` [347] are some of the most frequently used tools.

Fig. 8 depicts the most frequently used languages in experimental studies. It can be seen that mutation is mostly applied at the code/implementation level with the respective test subjects implemented in Java and C. Other commonly used programming languages that are used in mutation experiments include JavaScript and AspectJ. Finally, mutation has also been applied to other test subjects such as security protocols [316], feature models [272] and regular expressions [111], which are distributed in the remaining categories of the figure.

[b] Test prioritization techniques are typically assessed based on Average Percentage Faults Detected (APFD) or Normalized APFD (NAPFD) [392].

[c] The figure does not include tools that were used in fewer than four publications.

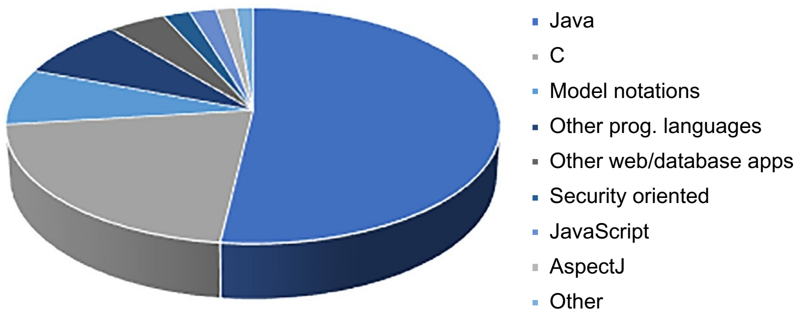


Fig. 8 Experimental studies: targeted artifacts.

Since mutation testing is increasingly used in experimental studies, potential issues with this practice can have serious implications on many research studies. Although there is some empirical evidence suggesting that mutants behave like real faults [41–43], these are only preliminary results, contradicted by other studies [63,393], and can be questioned if not suitable experimental care is taken. Unfortunately, recent research has shown that mutation testing is vulnerable to a number of confounding factors, such as those discussed in this section, that researchers should be aware and cater for. The influence of these factors can be severe and lead to questioning many findings of empirical research. In the remainder of this section, we discuss the influential factors that can bias the results of mutation-based test assessment and how we can mitigate them.

9.2 Programming Language and Mutant Operators

One of the main factors influencing the effectiveness of mutation-based test assessment is the programming language and the mutants that are used in the experimental study [33,393,394]. Applying mutation testing requires defining mutants based on the language's constructs. Thus, it is likely that we are not able to use the same mutants for different languages. Additionally, languages following different typing disciplines and programming paradigms may also require different sets of mutant operators. Certain types of mutants might be more effective in one paradigm than another. For instance, strongly typed languages may produce fewer mutants than weakly typed ones [152]. Similarly, object oriented code tends to have simpler method functionality, but more complex interactions among the methods (or classes) than imperative (procedural) languages [395]. Therefore, mutants encoding intramethod faults [9] might not be effective. Another example is the Java language runtime checking, which may result in mutants that are easier to kill than those in C.

Namin and Kakarla [393] demonstrated that the correlation between mutants and faults differs significantly across different programming languages. Kintis et al. [152] and Baker and Habli [57] also report significant differences between mutants of different languages, in particular between C—Java and C—Ada, respectively. Similarly, there are significant difference across different types of mutant operators, as reported by Namin and Kakarla [393]. Kurtz et al. [122] showed that it is hard to select mutant operator sets that perform similarly well on different programs. Therefore, when using mutation testing, *it is important to carefully select mutant operators that are appropriate to the programming language studied.*

9.3 Subsumed Mutant Threat

A major concern when using mutation testing is related to the "quality" of the employed mutants. In case the mutants we are using are trivial, then we only measure the ability of test suites to cover some parts of the code, instead of their ability to uncover faults at these parts. This problem is called the "subsumed mutant threat" [36]. The problem becomes particularly important when we have large number of redundant mutants. Unfortunately, when assessing testing methods, one test technique might achieve a significant advantage over another by killing redundant than nonredundant mutants. This would be a case of inadequately scientific methodology leading to possible incorrect scientific conclusions.

Recent research has shown that redundant mutants tend to skew the mutation score measurement leading to serious threats to the validity of empirical research. Andrews et al. [42] noted that the difficulty of revealing faults and killing mutants may influence the experimental results. Thus, they pointed out that it may be important to filter out the subset of trivial mutants in order to set a representative relation between mutants and real faults. Visser [396] suggested controlling for mutants' reachability in order to identify mutants that are hard to kill (hard to infect and propagate). Papadakis et al. [36] used the notion of mutant subsumption, demonstrating empirically that there is a very good chance (estimated to be more than 60% for arbitrary experiments) to compromise scientific conclusions, due to this subsumed mutant thread [36]. Similarly, Kurtz et al. [122] replicated previous studies on selective mutation and found that they perform well when considering redundant mutants but perform poorly when discarding them.

There are many studies advocating some form of "refined" mutation score for mitigating the problems caused by mutant redundancies. The first study attempting to address this problem was that of Kintis et al. [39] who suggested using disjoint mutants, i.e., a small representative subset of all mutants in order to remove all the mutant redundancies from the set of mutants that is used for test assessment. Consider that we have a set of *N* mutants, a representative subset, say *D*, means that any test suite that kills this subset of mutants also kills the *N* mutants. No redundancy between the mutants of *D* means that it is not safe to remove any mutant from this set because in this case we fail to kill all the *N* mutants.

Computing the true disjoint mutant set is impossible and thus, in the context of controlled experiments, it is approximated by a test suite. This dynamic approximation of the disjoint mutants can be computed using the Algorithm 1 [36,167]. In this algorithm, the live and duplicate mutants are removed first (from *S*, lines 2 and 3). Then, the mutant that is joint (subsuming) with the highest number of live mutants is selected (lines 8–15). This is the mutant that it is killed by test cases, which manage to collaterally kill the highest number of other mutants. This mutant is then added to the disjoint set *D* (line 16) and the joint mutants are removed from *S* (line 17). This process is repeated until *S* is empty. Finally, the set of disjoint mutants, *D*, is returned.

Other studies of redundancy reduction include that of Kaminski et al. [76,163] and Just et al. [164], which suggested removing some instances of the relational and logical mutant operators, in order to improve the accuracy of the mutation score. Ammann et al. [397] introduced the notion of minimal mutants (smallest possible set of mutants)[d] and Kurtz et al. [398] suggested selecting the minimal sets of mutants using mutant subsumption graphs. In another study, Kurtz et al. [169] proposed using symbolic execution to approximate subsuming mutants. Papadakis et al. [37] and Kintis et al. [152] suggested using compiler optimizations to remove duplicated mutants (a special form of redundant mutants) as a way to strengthen experimental rigor.

Overall, all these studies found that a large percentage of mutants are redundant, indicating potential inflation problems for the studies that have not take account of redundancy. Kintis et al. [39] report that disjoint mutants were approximately 9% of all the mutants, Ammann et al. [397] that minimal mutants were 1.2%, and Kurtz et al. [398] that they were 4%. These results

[d] The difference between the notions of minimal and disjoint mutants is that minimal mutants is the smallest possible representative set of mutants while the disjoint ones is a set that has no redundancies (perhaps not the minimal one) [33,39].

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
$D = \varnothing$
                                        /* Remove live mutants */
$S = S \setminus \{m \in S \mid \forall i \in 1..|T|, M_{ij} \neq 1\}$
                                        /* Remove duplicate mutants */
$S = S \setminus \{m \in S \mid \exists m' \in S \mid \forall i \in 1..|T|, M_{ij(m)} = M_{ij(m')}\}$
while ($|S| > 0$) do
    maxJoint = 0
    jointMut = null
    maxMutDisjoint = null
                                        /* Select the most disjoint mutant */
    foreach ($m \in S$) do
        sub$_m = \{m' \in S | \forall i \in 1..|T|, (M_{ij(m)} = 1) \Rightarrow (M_{ij(m')} = 1)\}$
        if ($|\text{sub}_m| >$ maxJoint) then
            maxJoint = $|\text{sub}_m|$
            maxMutDisjoint = $m$
            jointMut = sub$_m$
        end
    end
                                        /* Add the most disjoint mutant to $D$ */
    $D = D \cup \{\text{maxMutDisjoint}\}$
                                        /* Remove the joint mutants from the remaining */
    $S = S \setminus$jointMut
end
return $D$
```

motivated the work of Papadakis et al. [36] that found that redundant mutants can bias experimental results (approximately 60% for arbitrary experiments). Therefore, researchers should *identify and discard as many subsumed mutants as possible before conducting any test assessment.*

9.4 Test Suite Strength and Size

Failure to account for test suite strength can also adversely affect the scientific findings of empirical studies. Chekam et al. [6], studied the relation between faults and mutants, report that low-strength test suites are vulnerable to noise effects "two studies with below-threshold coverage may yield different findings, even when the experimenters follow identical experimental procedures." Thus, their study concluded that test suite strength plays a central role when conducting an experiment.

In particular, the study of Chekam et al. [6] showed that there is no practical difference between test criteria when relatively low-strength test suites are used. By contradiction, higher-strength test suites yield larger differences for test criteria. This is particularly important, because it indicates that empirical studies need to improve the strength of their test suites before conducting the experiment. Unfortunately, the mutation strength (over other test techniques) is only observable using strong test suites. For instance, one might conclude that a test technique or criterion is ineffective (compared to another), while in fact it is not, simply because the superiority of this criterion is only observable using stronger test suites.

Recent studies have also identified that test suite size (number of test cases) introduces another confounding factor that should be brought under experimental control [61]. Going a step further, Namin and Kakarla [393] observed that by using different test suite sizes, an experimenter will observe different correlations between faults and mutants. Since test suite size can be considered as a proxy measure of test suite's strength, this finding reconfirms the findings of Chekam et al. [6], suggesting experiments should consider both test suite size and test suite strength.

Another source of variation of empirical results is due to selection of candidate test cases. Consider the case where we want to compare two test techniques. In this case, we need two test suites, one that simulates the result of the first technique and one that simulates the result of the second technique. This is usually performed by randomly sampling of test cases from a test pool or by randomly generating test suites. Thus, two sets of test cases are to be compared. A problem typically arises in this scenario is that these two test suites need to adequately simulate the results of the techniques. Therefore, the two sets need to be free from redundant test cases [33,167,397], which might otherwise inadvertently bias experimental results. An easy way to ameliorate this problem is to select test cases that only increase coverage, while discarding all the others [33,42].

Another concern derives from the test case selection. As this may be stochastic, it is likely that different selections of test cases (at random) may result in different results; perhaps very different results if the experimenter happens to be unlucky. To reduce this problem, researchers usually select multiple sets of test cases and perform an inferential statistical analysis on the set of results as a whole [399,400]. Delamaro and Offutt [401] investigated the influence of using multiple sets of test cases (selected randomly) and found that "averaging over multiple programs was effective in reducing the variance in the mutation scores introduced by specific tests." Therefore, they found that in case it is too expensive to perform multiple repeated experiments a single test set (per program) over a relatively large number of subject can be enough to provide accurate average values.

Overall, researchers are advised to *carefully select their test suites*. Depending on the evaluation scenario it might be important to *control for test suite size* and *reduce redundant test cases*.

9.5 Mutation Testing Tools

An often-ignored parameter that can also inadvertently bias experimental findings relates to the choice of mutation testing tool. Mutation testing tools implement different operators and have different implementation details, most of which can influence the experimental outcome [394]. As already explained, the choice of mutant operators affects significantly the results of an experiment. However, different implementations of the same operators are likely to produce different mutants and merely provide divergent results.

The studies of Kintis et al. [394,402] and Gopinath et al. [403] demonstrate that there is a large degree of disagreement between the judgements made by the most popular Java mutation testing tools. The studies of Kintis et al. [394] and Marki and Lindström [404] cross evaluate the Java mutation testing tools and identify specific implementation weaknesses. This motivated the work of Laurent et al. [33,167], who compared Java mutation testing tools and implemented one (called $\mathrm{PIT}_{\mathrm{RV}}$ [147]) that is "at least as strong as the mutants of all the other tools together." Unfortunately, these studies are only concerned with Java so there is no clear evidence concerning the C mutation testing tools (or tools for other less widely studied languages). Taken together, these results suggest that the *choice of a mutation tool need to be carefully introduced and justified* in best practice empirical studies.

9.6 Clean Program Assumption

Mutation-based test assessment can be viewed as a simulation that involves two "roles"; the faults role (played by the mutants) and the "oracle" role (played by the original program). By aligning this simulation to the reality, we can say that developers produce the faulty programs (simulated by the mutants) which they test using a test oracle (simulated by the original program).

Naturally, testers apply their tools and techniques on the mutant program versions, check whether they can find any unexpected behavior, as defined by the test oracle and report on any bugs found.

As intuitive as this seems, the practice of test assessment is performed differently. It is a common practice to apply tools and test techniques on the original program and then check their fault revealing power by executing tests on the mutants. This practice may be less time-consuming but it makes an implicit assumption that coverage measurements (or the application of test techniques) on the original program are representative (or very similar) of those on the mutant programs. This assumption is called the "clean program assumption" (CPA) [6]. The assumption can be problematic since test suites are assessed on the mutant program versions instead of the original program from which (and for which) they were applied.

Unfortunately, Chekam et al. [6] demonstrated that the CPA does not hold and therefore cannot be relied upon. The study also showed that CPA has the potential of changing the outcome of empirical studies if not brought under experimental control. Overall, the Chekam et al. revisited previous empirical questions concerning the usefulness of test adequacy criteria, using a robust methodology that accounts for CPA and showed that mutation testing outperform statement and branch coverage for real fault revelation. These results suggest that experiments dealing with the real fault revelation question, should *report on the CPA*. If it is not possible to take CPA into account (potentially due to execution cost), researchers are advised to report the amount of time required by the performed study.

10. A SEVEN-POINT CHECK LIST OF BEST PRACTICES ON USING MUTATION TESTING IN CONTROLLED EXPERIMENTS

The fundamental experimental factors surveyed in Section 9 highlight the many pitfalls that can compromise or even invalidate the scientific findings and conclusions of a controlled experiment that uses mutation testing. It can be a daunting challenge for experimenters and researchers to be sure they have catered for all of the potential threats to validity that have accrued over four decades of literature recording the development of mutation testing.

Therefore, to address this challenge, in this section, we provide a simple seven-step checklist that aims to give experimenters the confidence that they are compliant with best practice reporting of results. Ensuring that all seven steps are met is relatively straightforward, because it simply involves explaining and justifying choices that may affect conclusion validity. Nevertheless, experimenters who follow these seven steps help other researchers replicate and investigate, properly, the influence of such potentially confounding factors, thereby contributing to the overall experimental robustness of their study.

1. *Mutant selection*: Explain the choice of mutant operators. One of the most important things that experimenters need to explain is the appropriateness of the chosen mutant operators with respect to the programming language used.
2. *Mutation testing tool*: Justify the choice of mutation testing tool. The choice of mutation testing tool needs to be made carefully as at the current state, mutation testing tools differ significantly [394,402]. To support the reproducibility and comprehension of the experimental results, researchers should also clearly describe the exact version of the employed mutation testing tool. If the used tool is not a publicly available, researchers should list the exact transformation rules (mutant instances supported by each operator [394,402]) that are supported by the mutant operators selected. Unfortunately, our survey found that more than a quarter of the empirical studies does not report such details. The objective is to provide readers with the low-level details that might vary from one study to another, so that these can be accounted for in subsequent studies.
3. *Mutant redundancy*: Justify the steps taken to control mutant redundancy. As we discussed in Section 9.3, mutant redundancy may have a large impact on the validity of the assessment. Therefore, it is important to explain how mutant redundancy is handled (perhaps in the threats to validity section). Where possible, experimenters are advised to additionally use techniques such as TCE [152] to remove the duplicate mutants (in case the interest is on the achieved score of a technique), or a dynamic approximation of the disjoint mutation score [33,36] (in case the interest is on comparing test techniques). As already discussed, the approximation of the disjoint mutants can be made by using Algorithm 1. In case these techniques are expensive, researchers are advised to clarify this and contrast their findings on a (small) sample of cases where mutant redundancy is controlled.
4. *Test suite choice and size*: Explain the choice of test suite and any steps taken to account for the effects of test suite size, where appropriate. Ideally, an experimenter would like to have large, diverse (i.e., mutants are killed by multiple test cases), and high-strength (i.e., killing the

majority of the mutants) test suites. As such test suites are rare in most of the open source projects, researchers are advised to demonstrate and contrast their findings with a (small) sample of subjects with strong and diverse test suites (perhaps in addition to the chosen subjects). Alternatively, experimenters may consider using automated tools to augment their test suites. Overall, the objective is to allow other researchers to create a similar test suite and/or to experiment with different choices of suite and measure the effects of such choices.

5. *Clean program assumption*: Explain what the study relies on the CPA assumption. Ideally, where possible, the CPA should not be relied upon; testing should be applied to the faulty programs (instead of the clean, nonfaulty ones). If this is not possible (potentially due to execution cost or lack of resources), researchers are advised to note the reliance on the CPA. Its effects may be small in some cases, justifying reliance on this assumption. Either way, explicitly stating whether or not it is relied upon will aid clarity and facilitate subsequent studies.
6. *Multiple experimental repetitions*: Clarify the number of experimental repetitions. Ideally, when techniques make stochastic choices they should be assessed by multiple experimental repetitions [400,405]. In practice, this might not be possible due to the required execution time or other constraints. In this case, researchers have to choose between experiments with many subjects but few repetitions or experiments with few subjects and many repetitions; research suggests that it is preferable to choose the second option [401]. Of course, this choice needs to be clarified according to the specific context and goals of the study.
7. *Presentation of the results*: Clarify the granularity level of the empirical results. Many empirical studies compute mutation scores over the whole subject projects they are using (one score per project). Since, this practice may not generalize to other granularity levels[e] (such as unit level) [167], researchers should report and explain the suitability of the chosen granularity level at the given application context.

11. CONCLUSION AND FUTURE DIRECTIONS

This chapter surveyed the recent trends and advances on mutation testing. It offers a concise description of the mutation testing problems, methods, applications, and best practices for applying mutation testing

[e] Two methods can have a similar number of mutants killed on a project (overall number), but quite different numbers, of mutants killed, on the individual units of the project.

(either as a test technique or as an experimental methodology). Based on the data we collected, we demonstrate that there is a growing interest in the subject. Interestingly, even 8 years after the first observation of this trend, by Jia and Harman [27], the interest in the field is still increasing markedly.

The interest in the field is related to both fundamental research advances and practical applications such as tool support and use in controlled experiments. Our analysis shows that many tools and techniques have been introduced these last 10 years. Many of these advances are already widely used by researchers. At the same time major companies report that they experiment with mutation in order to include in their practices. Hopefully, practitioners will soon use mutation as well. All these observations may be seen as evidence supporting the claim that mutation testing is reaching a state of maturity. In summary, the research interest in mutation testing is divided into the following general categories:

- Solutions to the problems of mutation analysis (fundamental advances of mutation).
- Mutation applied on new languages and artifacts. New mutation testing tools also appear.
- Use of mutants as a means to support other software engineering activities (e.g., fault localization [17]).
- Use of mutation testing advances to support controlled experiments.

Recent work in the area focuses on building scalable and practical tools that can push mutation testing toward industry and everyday use. The rest of this section is dedicated to summarize the mutation testing open problems, barriers, and areas that we believe will attract attention in the near future.

11.1 Open Problems

One of the main open problems of mutation regards the detection of the equivalent and redundant mutants. As we already discussed, there are many techniques tackling this problem, either directly or indirectly, but unfortunately the problem remains largely unresolved. Overall, the current research results show that only few of the mutants produced (approximately 5%) is practically useful. The rest is noise to the process with severe consequences [36].

Overall, mutation testing requires models that will guide the mutations toward small semantic deviations that are in a sense disjoint, instead of blind syntactical mutations. Unfortunately, there is no clear theory or consensus on which types and instances of mutants we should use. Some initial results indicate that almost all the mutant operators are of some value. The fact that most of the existing tools are limited to a small number of mutant operators is restrictive and to some extend arbitrary. Thus, in future, mutation may be tailored toward few diverse and "useful" mutants that bring value to the tester (regardless of the operators used) [41].

The lack of clear theory on which mutants are of some value has restricted most of the previous research on first-order mutation. Higher order mutation appears to have similar characteristics with the first-order mutation as it produces subtle mutants. Of course the great majority of them are redundant, but in theory a smart mutant selection process can identify them. Therefore, future mutation may identify ways to generate and use those valuable higher order mutants.

Another important aspect concerns the automatic mutation-based generation of test cases and test oracles. Although the last 10 years there are major advances on this area of research, the problem remains. Most of the automated approaches fail to kill a substantial number of mutants and recent empirical evaluations show that automatic test generation techniques fail to cover most of the critical program areas. Therefore, there is little work on improving test suites using mutants. Perhaps this is attributed to the lack of understanding and modeling of the error propagation. Recent research has shown that failed mutant propagation is the basic ingredient that makes mutation testing powerful [6]. Much work remains to be done until we can automatically produce high quality test cases through high quality mutants.

Although researchers have identified mutation as a strong test criterion, there is neither clear understanding nor much empirical evidence concerning whether and when mutants are correlated with real faults. What types of faults are not captured by simple or complex mutants? What percentage of future regression errors can we capture with mutations? When is it appropriate to stop the testing process? How should we integrate mutation testing into our development process? Of course these questions need to be answered under the light of specific development paradigms and application domains. These are open questions, hopefully to be answered by future research.

Model-based mutation is one of the areas that has not been researched much (compared to code-based mutation) over the last years. Despite this, we see a growing interest toward this direction. There is a recent dedicated survey on this subject [30] and multiple high profile publications over the last couple of years. Additionally, very recently efficient and scalable tools have been built, e.g., the VIBeS tool [12], which hopefully will push the research in this area further.

Finally, there are many new areas of research that can benefit from the use of mutants. The current trend is to explore the behavior space of mutants, instead of the original program, to identify several interesting aspects, either functional or nonfunctional. Thus, the conformance of models, the generation and improvement of models, the improvements of program security, and debugging activities are only a few examples where mutants have been shown to be spectacularly useful and effective. Future research is heading toward this line of research with many new and exciting applications of mutation analysis.

REFERENCES

[1] J. Offutt, A mutation carol: past, present and future, Inf. Softw. Technol. 53 (10) (2011) 1098–1107, https://doi.org/10.1016/j.infsof.2011.03.007.

[2] P.E. Black, V. Okun, Y. Yesha, Mutation operators for specifications, in: The Fifteenth IEEE International Conference on Automated Software Engineering, ASE 2000, Grenoble, France, September 11–15, 2000, 2000, p. 81, https://doi.org/10.1109/ASE.2000.873653.

[3] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P. Schobbens, P. Heymans, Automata language equivalence vs. simulations for model-based mutant equivalence: an empirical evaluation, in: 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13–17, 2017, 2017, pp. 424–429, https://doi.org/10.1109/ICST.2017.46.

[4] M.E. Delamaro, J.C. Maldonado, A.P. Mathur, Interface mutation: an approach for integration testing, IEEE Trans. Softw. Eng. 27 (3) (2001) 228–247, https://doi.org/10.1109/32.910859.

[5] P.R. Mateo, M.P. Usaola, J. Offutt, Mutation at system and functional levels, in: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7–9, 2010, Workshops Proceedings, 2010, pp. 110–119, https://doi.org/10.1109/ICSTW.2010.18.

[6] T.T. Chekam, M. Papadakis, Y.L. Traon, M. Harman, An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption, in: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017, 2017, pp, 597–608. http://dl.acm.org/citation.cfm?id=3097440.

[7] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, J.J. Domínguez-Jiménez, Assessment of class mutation operators for C++ with the MuCPP mutation system, Inf. Softw. Technol. 81 (2017) 169–184, https://doi.org/10.1016/j.infsof.2016.07.002.

[8] A. Derezinska, K. Kowalski, Object-oriented mutation applied in common intermediate language programs originated from C#, in: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, March 21–25, 2011, Workshop Proceedings, 2011, pp. 342–350, https://doi.org/10.1109/ICSTW.2011.54.

[9] Y. Ma, J. Offutt, Y.R. Kwon, MuJava: an automated class mutation system, Softw. Test. Verif. Reliab. 15 (2) (2005) 97–133, https://doi.org/10.1002/stvr.308.

[10] S. Mirshokraie, A. Mesbah, K. Pattabiraman, Efficient JavaScript mutation testing, in: Sixth IEEE International Conference on Software Testing, Verification and

Validation, ICST 2013, Luxembourg, Luxembourg, March 18–22, 2013, 2013, pp. 74–83, https://doi.org/10.1109/ICST.2013.23.

[11] N. Li, M. West, A. Escalona, V.H.S. Durelli, Mutation testing in practice using Ruby, in: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13–17, 2015, 2015, pp. 1–6, https://doi.org/10.1109/ICSTW.2015.7107453.

[12] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P. Schobbens, P. Heymans, Featured model-based mutation analysis, in: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016, 2016, pp. 655–666, https://doi.org/10.1145/2884781.2884821.

[13] Y. Ma, Y.R. Kwon, J. Offutt, Inter-class mutation operators for Java, in: 13th International Symposium on Software Reliability Engineering (ISSRE 2002), November 12–15, 2002, Annapolis, MD, USA, 2002, pp. 352–366, https://doi.org/10.1109/ISSRE.2002.1173287.

[14] D. Le, M.A. Alipour, R. Gopinath, A. Groce, MuCheck: an extensible tool for mutation testing of haskell programs, in: International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA, July 21–26, 2014, 2014, pp. 429–432, https://doi.org/10.1145/2610384.2628052.

[15] E. Omar, S. Ghosh, An exploratory study of higher order mutation testing in aspect-oriented programming, in: 23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27–30, 2012, 2012, pp. 1–10, https://doi.org/10.1109/ISSRE.2012.6.

[16] J. Tuya, M.J.S. Cabal, C. de la Riva, Mutating database queries, Inf. Softw. Technol. 49 (4) (2007) 398–417, https://doi.org/10.1016/j.infsof.2006.06.009.

[17] M. Papadakis, Y.L. Traon, Metallaxis-FL: mutation-based fault localization, Softw. Test. Verif. Reliab. 25 (5–7) (2015) 605–628, https://doi.org/10.1002/stvr.1509.

[18] C. Le Goues, T. Nguyen, S. Forrest, W. Weimer, GenProg: a generic method for automatic software repair, IEEE Trans. Softw. Eng. 38 (1) (2012) 54–72, https://doi.org/10.1109/TSE.2011.104.

[19] T. Loise, X. Devroey, G. Perrouin, M. Papadakis, P. Heymans, Towards security-aware mutation testing, in: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13–17, 2017, 2017, pp. 97–102, https://doi.org/10.1109/ICSTW.2017.24.

[20] Y. Jia, F. Wu, M. Harman, J. Krinke, Genetic improvement using higher order mutation, in: Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11–15, 2015, Companion Material Proceedings, 2015, pp. 803–804, https://doi.org/10.1145/2739482.2768417.

[21] W.B. Langdon, B.Y.H. Lam, M. Modat, J. Petke, M. Harman, Genetic improvement of GPU software, Genet. Program. Evolvable Mach. 18 (1) (2017) 5–44, https://doi.org/10.1007/s10710-016-9273-9.

[22] R.G. Hamlet, Testing programs with the aid of a compiler, IEEE Trans, Softw. Eng. 3 (4) (1977) 279–290.

[23] R.A. DeMillo, R.J. Lipton, F.G. Sayward, Program mutation: a new approach to program testing, in: Infotech State of the Art Report, Software Testing, vol, 2, 1979, pp. 107–126.

[24] R.A. DeMillo, R.J. Lipton, F.G. Sayward, Hints on test data selection: help for the practicing programmer, IEEE Comput. 11 (4) (1978) 34–41, https://doi.org/10.1109/C-M.1978.218136.

[25] R.A. DeMillo, Test adequacy and program mutation, in: Proceedings of the 11th International Conference on Software Engineering, Pittsburg, PA, USA, May 15–18, 1989, 1989, pp. 355–356, https://doi.org/10.1145/74587.74634.

[26] A.J. Offutt, R. Untch, Mutation 2000: uniting the orthogonal, in: W,E. Wong (Ed.), Mutation 2000, Kluwer, San Jose, CA, USA, 2001, ISBN: 0-7923-7323-5, pp. 45–55.
[27] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Trans. Softw. Eng. 37 (5) (2011) 649–678, https://doi.org/10.1109/TSE.2010.62.
[28] L. Madeyski, W. Orzeszyna, R. Torkar, M. Jozala, Overcoming the equivalent mutant problem: a systematic literature review and a comparative experiment of second order mutation, IEEE Trans. Softw. Eng. 40 (1) (2014) 23–42, https://doi.org/10.1109/TSE.2013.44.
[29] F.C. Souza, M. Papadakis, V.H.S. Durelli, M.E. Delamaro, Test data generation techniques for mutation testing: a systematic mapping, in: Proceedings of the 11th ESELAW, 2014, pp, 1–14.
[30] F. Belli, C.J. Budnik, A. Hollmann, T. Tuglular, W.E. Wong, Model-based mutation testing—approach and case studies, Sci. Comput. Program. 120 (2016) 25–48, https://doi.org/10.1016/j.scico.2016.01.003.
[31] R.A. Silva, S. do Rocio Senger de Souza, P.S.L. de Souza, A systematic review on search based mutation testing, Inf. Softw. Technol. 81 (2017) 19–35, https://doi.org/10.1016/j.infsof.2016.01.017.
[32] P. Ammann, J. Offutt, Introduction to Software Testing, second ed,, Cambridge University Press, 2016. ISBN: 978-1-107-17201-2.
[33] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y.L. Traon, A. Ventresque, Assessing and improving the mutation testing practice of PIT, in: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2017, pp. 430–435, https://doi.org/10.1109/ICST.2017.47.
[34] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, C. Zapf, An experimental determination of sufficient mutant operators, ACM Trans. Softw. Eng. Methodol. 5 (2) (1996) 99–118, https://doi.org/10.1145/227607.227610.
[35] J.B. Goodenough, S.L. Gerhart, Toward a theory of test data selection, IEEE Trans. Softw. Eng. 1 (2) (1975) 156–173, https://doi.org/10.1109/TSE.1975.6312836.
[36] M. Papadakis, C. Henard, M. Harman, Y. Jia, Y.L. Traon, Threats to the validity of mutation-based test assessment, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18–20, 2016, 2016, pp. 354–365, https://doi.org/10.1145/2931037.2931040.
[37] M. Papadakis, Y. Jia, M. Harman, 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, vol. 1, 2015, pp. 936–946, https://doi.org/10.1109/ICSE.2015.103.
[38] Y. Jia, M. Harman, Higher order mutation testing, Inf. Softw. Technol. 51 (10) (2009) 1379–1393, https://doi.org/10.1016/j.infsof.2009.04.016.
[39] M. Kintis, M. Papadakis, N. Malevris, Evaluating mutation testing alternatives: a collateral experiment, in: 17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30–December 3, 2010, 2010, pp. 300–309, https://doi.org/10.1109/APSEC.2010.42.
[40] Karl Popper, https://en.wikiquote.org/wiki/Karl_Popper (accessed 06.10.2017).
[41] M. Papadakis, D. Shin, S. Yoo, D. Bae, Are mutation scores correlated with real fault detection? A large scale empirical study on the relationship between mutants and real faults, in: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27–3 June, 2018, 2018,
[42] J.H. Andrews, L.C. Briand, Y. Labiche, A.S. Namin, Using mutation analysis for assessing and comparing testing coverage criteria, IEEE Trans. Softw. Eng. 32 (8) (2006) 608–624, https://doi.org/10.1109/TSE.2006.83.

[43] R. Just, D. Jalali, L. Inozemtseva, M.D. Ernst, R. Holmes, G. Fraser, Are mutants a valid substitute for real faults in software testing? 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, 654–665, https://doi.org/10.1145/2635868.2635929.

[44] P.G. Frankl, S.N. Weiss, An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria, in: Symposium on Testing, Analysis, and Verification, 1991, pp. 154–164, https://doi.org/10.1145/120807.120821.

[45] P.G. Frankl, S.N. Weiss, An experimental comparison of the effectiveness of branch testing and data flow testing, IEEE Trans. Softw. Eng. 19 (8) (1993) 774–787, https://doi.org/10.1109/32.238581.

[46] A.J. Offutt, J. Pan, K. Tewary, T. Zhang, An experimental evaluation of data flow and mutation testing, Softw. Pract. Exp. 26 (2) (1996) 165–176, https://doi.org/10.1002/(SICI)1097-024X(199602)26:2⟨165::AID-SPE5⟩3.0.CO;2-K.

[47] P.G. Frankl, S.N. Weiss, C. Hu, All-uses vs mutation testing: an experimental comparison of effectiveness, J. Syst. Softw. 38 (3) (1997) 235–253, https://doi.org/10.1016/S0164-1212(96)00154-9.

[48] P.G. Frankl, O. Iakounenko, Further empirical studies of test effectiveness, in: SIGSOFT 1998, Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lake Buena Vista, FL, USA, November 3–5, 1998, 1998, pp. 153–162, https://doi.org/10.1145/288195.288298.

[49] L.C. Briand, D. Pfahl, Using simulation for assessing the real impact of test-coverage on defect-coverage, IEEE Trans. Reliab. 49 (1) (2000) 60–70, https://doi.org/10.1109/24.855537.

[50] M. Chen, M.R. Lyu, W.E. Wong, Effect of code coverage on software reliability measurement, IEEE Trans. Reliab. 50 (2) (2001) 165–170, https://doi.org/10.1109/24.963124.

[51] A.S. Namin, J.H. Andrews, The influence of size and coverage on test suite effectiveness, in: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19–23, 2009, 2009, pp. 57–68, https://doi.org/10.1145/1572272.1572280.

[52] N. Li, U. Praphamontripong, J. Offutt, An experimental comparison of four unit test criteria: mutation, edge-pair, all-uses and prime path coverage, in: Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, CO, USA, April 1–4, 2009, Workshops Proceedings, 2009, pp. 220–229, https://doi.org/10.1109/ICSTW.2009.30.

[53] M. Papadakis, 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, https://doi.org/10.1109/ICSTW.2010.50.

[54] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, B. Meyer, On the number and nature of faults found by random testing, Softw. Test. Verif. Reliab. 21 (1) (2011) 3–28, https://doi.org/10.1002/stvr.415.

[55] S. Kakarla, S. Momotaz, A.S. Namin, An evaluation of mutation and data-flow testing: a meta-analysis, in: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, March 21–25, 2011, Workshop Proceedings, 2011, pp. 366–375, https://doi.org/10.1109/ICSTW.2011.51.

[56] Y. Wei, B. Meyer, M. Oriol, Is branch coverage a good measure of testing effectiveness? in: Springer, Berlin, Heidelberg, 2012, ISBN: 978-3-642-25231-0, pp. 194–212, https://doi.org/10.1007/978-3-642-25231-0_5.

[57] R. Baker, I. Habli, An empirical evaluation of mutation testing for improving the test quality of safety-critical software, IEEE Trans. Softw. Eng. 39 (6) (2013) 787–805, https://doi.org/10.1109/TSE.2012.56.

[58] M.M. Hassan, J.H. Andrews, Comparing multi-point stride coverage and dataflow coverage, in: 35th International Conference on Software Engineering, ICSE 2013, San Francisco, CA, USA, May 18–26, 2013, 2013, pp. 172–181, https://doi.org/10.1109/ICSE.2013.6606563.

[59] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M.A. Alipour, D. Marinov, Comparing non-adequate test suites using coverage criteria, in: International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15–20, 2013, 2013, pp. 302–313, https://doi.org/10.1145/2483760.2483769.

[60] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M.A. Alipour, D. Marinov, Guidelines for coverage-based comparisons of non-adequate test suites, ACM Trans. Softw. Eng. Methodol. 24 (4) (2015) 22:1–22:33, https://doi.org/10.1145/2660767.

[61] L. Inozemtseva, 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 7, 2014, 2014, pp. 435–445, https://doi.org/10.1145/2568225.2568271.

[62] R. Gopinath, C. Jensen, A. Groce, Code coverage for suite evaluation by developers, in: 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India, May 31–June 7, 2014, 2014, pp. 72–82, https://doi.org/10.1145/2568225.2568278.

[63] I. Ahmed, R. Gopinath, C. Brindescu, A. Groce, C. Jensen, Can testedness be effectively measured? in: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13–18, 2016, 2016, pp, 547–558, https://doi.org/10.1145/2950290.2950324.

[64] R. Ramler, T. Wetzlmaier, C. Klammer, An empirical study on the application of mutation testing for a safety-critical industrial software system, in: Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3–7, 2017, 2017, pp. 1401–1408, https://doi.org/10.1145/3019612.3019830.

[65] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, L. Ouarbya, Formalizing executable dynamic and forward slicing, in: 4th International Workshop on Source Code Analysis and Manipulation (SCAM 04), 2004, pp, 43–52. Los Alamitos, CA, USA.

[66] K.H. Brodersen, F. Gallusser, J. Koehler, N. Remy, S.L. Scott, Inferring causal impact using Bayesian structural time-series models, Ann, Appl. Stat. 9 (2015) 247–274.

[67] L.J. Morell, A theory of fault-based testing, IEEE Trans. Softw. Eng. 16 (8) (1990) 844–857, https://doi.org/10.1109/32.57623.

[68] A.J. Offutt, Investigations of the software testing coupling effect, ACM Trans. Softw. Eng. Methodol. 1 (1) (1992) 5–20, https://doi.org/10.1145/125489.125473.

[69] J. Voas, G. McGraw, Software Fault Injection: Inoculating Programs Against Errors, John Wiley & Sons, 1997, ISBN: 0-471-18381-4.

[70] R. Gopinath, C. Jensen, A. Groce, Mutations: how close are they to real faults? in: 25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3–6, 2014, 2014, pp, 189–200, https://doi.org/10.1109/ISSRE.2014.40.

[71] R. Gopinath, C. Jensen, A. Groce, The theory of composite faults, in: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2017, pp. 47–57, https://doi.org/10.1109/ICST.2017.12.

[72] W.B. Langdon, M. Harman, Y. Jia, Efficient multi-objective higher order mutation testing with genetic programming, J. Syst. Softw. 83 (12) (2010) 2416–2430, https://doi.org/10.1016/j.jss.2010.07.027.

[73] R. Geist, A.J. Offutt, F.C. Harris Jr, Estimation and enhancement of real-time software reliability through mutation analysis, IEEE Trans. Comput. 41 (5) (1992) 550–558, https://doi.org/10.1109/12.142681.

[74] I. Ahmed, C. Jensen, A. Groce, P.E. McKenney, Applying mutation analysis on kernel test suites: an experience report, in: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2017, pp. 110–115, https://doi.org/10.1109/ICSTW.2017.26.

[75] A.J. Offutt, S.D. Lee, An empirical evaluation of weak mutation, IEEE Trans. Softw. Eng. 20 (5) (1994) 337–344, https://doi.org/10.1109/32.286422.

[76] G. Kaminski, P. Ammann, J. Offutt, Improving logic-based testing, J. Syst. Softw. 86 (8) (2013) 2002–2012, https://doi.org/10.1016/j.jss.2012.08.024.

[77] P. Anbalagan, T. Xie, Automated generation of pointcut mutants for testing pointcuts in AspectJ programs, in: 19th International Symposium on Software Reliability Engineering (ISSRE 2008), November 11–14, 2008, Seattle/Redmond, WA, USA, 2008, pp. 239–248, https://doi.org/10.1109/ISSRE.2008.58.

[78] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, Mutation operators for WS-BPEL 2.0, in: 21th International Conference on Software & Systems Engineering and their Applications, 2008,

[79] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, Quantitative evaluation of mutation operators for WS-BPEL compositions, in: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7–9, 2010, Workshops Proceedings, 2010, pp. 142–150, https://doi.org/10.1109/ICSTW.2010.36.

[80] J. Boubeta-Puig, I. Medina-Bulo, A. García-Domínguez, Analogies and differences between mutation operators for WS-BPEL 2.0 and other languages, in: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, March 21–25, 2011, Workshop Proceedings, 2011, pp. 398–407, https://doi.org/10.1109/ICSTW.2011.52.

[81] S. Mirshokraie, A. Mesbah, K. Pattabiraman, Guided mutation testing for JavaScript web applications, IEEE Trans. Softw. Eng. 41 (5) (2015) 429–444, https://doi.org/10.1109/TSE.2014.2371458.

[82] P. Delgado-Pérez, S. Segura, I. Medina-Bulo, Assessment of C++ object-oriented mutation operators: a selective mutation approach, Softw. Test. Verif. Reliab. 27 (4–5) (2017) e1630. ISSN: 1099-1689, https://doi.org/10.1002/stvr.1630.

[83] J. Hu, N. Li, J. Offutt, An analysis of OO mutation operators, in: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, March 21–25, 2011, Workshop Proceedings, 2011, pp. 334–341, https://doi.org/10.1109/ICSTW.2011.47.

[84] F.C. Ferrari, J.C. Maldonado, A. Rashid, Mutation testing for aspect-oriented programs, in: First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9–11, 2008, 2008, pp. 52–61, https://doi.org/10.1109/ICST.2008.37.

[85] L. Bottaci, Type sensitive application of mutation operators for dynamically typed programs, in: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7–9, 2010, Workshops Proceedings, 2010, pp. 126–131, https://doi.org/10.1109/ICSTW.2010.56.

[86] M. Gligoric, S. Badame, R. Johnson, SMutant: a tool for type-sensitive mutation testing in a dynamic language, in: SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5–9, 2011, 2011, pp. 424–427, https://doi.org/10.1145/2025113.2025181.

[87] A.D.B. Alberto, A. Cavalcanti, M. Gaudel, A. Simão, Formal mutation testing for Circus, Inf. Softw. Technol. 81 (2017) 131–153, https://doi.org/10.1016/j.infsof.2016.04.003.

[88] U. Praphamontripong, J. Offutt, Applying mutation testing to web applications, in: Third International Conference on Software Testing, Verification and Validation,

ICST 2010, Paris, France, April 7–9, 2010, Workshops Proceedings, 2010, pp. 132–141, https://doi.org/10.1109/ICSTW.2010.38.
[89] U. Praphamontripong, J. Offutt, L. Deng, J. Gu, An experimental evaluation of web mutation operators, in: Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11–15, 2016, 2016, pp. 102–111, https://doi.org/10.1109/ICSTW.2016.17.
[90] U. Praphamontripong, J. Offutt, Finding redundancy in web mutation operators, in: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2017, pp. 134–142, https://doi.org/10.1109/ICSTW.2017.30.
[91] L. Deng, N. Mirzaei, P. Ammann, J. Offutt, Towards mutation analysis of Android apps, in: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13–17, 2015, 2015, pp. 1–10, https://doi.org/10.1109/ICSTW.2015.7107450.
[92] L. Deng, J. Offutt, P. Ammann, N. Mirzaei, Mutation operators for testing Android apps, Inf. Softw. Technol. 81 (2017) 154–168, https://doi.org/10.1016/j.infsof.2016.04.012.
[93] M.P. Usaola, G. Rojas, I. Rodríguez, S. Hernández, An architecture for the development of mutation operators, in: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2017, pp. 143–148, https://doi.org/10.1109/ICSTW.2017.31.
[94] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, D. Poshyvanyk, Enabling mutation testing for Android apps, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, 2017, ISBN: 978-1-4503-5105-8, pp. 233–244, https://doi.org/10.1145/3106237.3106275.
[95] R.A.P. Oliveira, E. Alégroth, Z. Gao, A.M. Memon, Definition and evaluation of mutation operators for GUI-level mutation analysis, in: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13–17, 2015, 2015, pp. 1–10, https://doi.org/10.1109/ICSTW.2015.7107457.
[96] V. Lelli, A. Blouin, B. Baudry, Classifying and qualifying GUI defects, CoRR (2017). http://arxiv.org/abs/1703.09567.
[97] R. Abraham, M. Erwig, Mutation Operators for Spreadsheets, IEEE Trans. Softw. Eng. 35 (1) (2009) 94–108, https://doi.org/10.1109/TSE.2008.73.
[98] H. Dan, R.M. Hierons, Semantic mutation analysis of floating-point comparison, in: Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17–21, 2012, 2012, pp. 290–299, https://doi.org/10.1109/ICST.2012.109.
[99] V. Jagannath, M. Gligoric, S. Lauterburg, D. Marinov, G. Agha, Mutation operators for actor systems, in: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7–9, 2010, Workshops Proceedings, 2010, pp. 157–162, https://doi.org/10.1109/ICSTW.2010.6.
[100] Y. Maezawa, K. Nishiura, H. Washizaki, S. Honiden, Validating ajax applications using a delay-based mutation technique, in: ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden, September 15–19, 2014, 2014, pp. 491–502, https://doi.org/10.1145/2642937.2642996.
[101] T. Xie, N. Tillmann, J. de Halleux, W. Schulte, Mutation analysis of parameterized unit tests, in: Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, CO, USA, April 1–4, 2009, Workshops Proceedings, 2009, pp. 177–181, https://doi.org/10.1109/ICSTW.2009.43.
[102] D.B. Brown, M. Vaughn, B. Liblit, T. Reps, The care and feeding of wild-caught mutants, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software

Engineering, ACM, New York, NY, USA, 2017, ISBN: 978-1-4503-5105-8, pp. 511–522, https://doi.org/10.1145/3106237.3106280.

[103] J. Nanavati, F. Wu, M. Harman, Y. Jia, J. Krinke, Mutation testing of memory-related operators, in: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13–17, 2015, 2015, pp. 1–10, https://doi.org/10.1109/ICSTW.2015.7107449.

[104] F. Wu, J. Nanavati, M. Harman, Y. Jia, J. Krinke, Memory mutation testing, Inf. Softw. Technol. 81 (2017) 97–111, https://doi.org/10.1016/j.infsof.2016.03.002.

[105] B.J. Garvin, M.B. Cohen, Feature interaction faults revisited: an exploratory study, in: IEEE 22nd International Symposium on Software Reliability Engineering, ISSRE 2011, Hiroshima, Japan, November 29–December 2, 2011, 2011, pp. 90–99, https://doi.org/10.1109/ISSRE.2011.25.

[106] M. Al-Hajjaji, F. Benduhn, T. Thüm, T. Leich, G. Saake, Mutation operators for preprocessor-based variability, in: Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, Salvador, Brazil, January 27–29, 2016, 2016, pp. 81–88, https://doi.org/10.1145/2866614.2866626.

[107] M.E. Delamaro, J. Offutt, P. Ammann, Designing deletion mutation operators, in: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014–April 4, 2014, Cleveland, OH, USA, 2014, pp. 11–20, https://doi.org/10.1109/ICST.2014.12.

[108] F. Belli, M. Beyazit, T. Takagi, Z. Furukawa, Mutation testing of "Go-Back" functions based on pushdown automata, in: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21–25, 2011, 2011, pp. 249–258, https://doi.org/10.1109/ICST.2011.30.

[109] R. Gopinath, E. Walkingshaw, How good are your types? Using mutation analysis to evaluate the effectiveness of type annotations, in: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2017, pp. 122–127, https://doi.org/10.1109/ICSTW.2017.28.

[110] R. Jabbarvand, S. Malek, muDroid: an energy-aware mutation testing framework for Android, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, 2017, ISBN: 978-1-4503-5105-8, pp. 208–219, https://doi.org/10.1145/3106237.3106244.

[111] P. Arcaini, A. Gargantini, E. Riccobene, MutRex: a mutation-based generator of fault detecting strings for regular expressions, in: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2017, pp. 87–96, https://doi.org/10.1109/ICSTW.2017.23.

[112] L. Zhang, S. Hou, J. Hu, T. Xie, H. Mei, Is operator-based mutant selection superior to random mutant selection? in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol, 1, ICSE 2010, Cape Town, South Africa, May 1–8, 2010, 2010, pp. 435–444, https://doi.org/10.1145/1806799.1806863.

[113] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, A. Groce, How hard does mutation analysis have to be, anyway? in: 26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2–5, 2015, 2015, pp, 216–227, https://doi.org/10.1109/ISSRE.2015.7381815.

[114] A.S. Namin, J.H. Andrews, D.J. Murdoch, Sufficient mutation operators for measuring test effectiveness, in: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10–18, 2008, 2008, pp. 351–360, https://doi.org/10.1145/1368088.1368136.

[115] M.E. Delamaro, L. Deng, V.H.S. Durelli, N. Li, J. Offutt, Experimental evaluation of SDL and one-op mutation for C, in: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014–April 4, 2014, Cleveland, OH, USA, 2014, pp. 203–212, https://doi.org/10.1109/ICST.2014.33.

[116] V.H.S. Durelli, N.M.D. Souza, M.E. Delamaro, Are deletion mutants easier to identify manually? in: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2017, pp, 149–158, https://doi.org/10.1109/ICSTW.2017.32.

[117] X. Yao, M. Harman, Y. Jia, A study of equivalent and stubborn mutation operators using human analysis of equivalence, in: 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India, May 31–June 7, 2014, 2014, pp. 919–930, https://doi.org/10.1145/2568225.2568265.

[118] J. Zhang, M. Zhu, D. Hao, L. Zhang, An empirical study on the scalability of selective mutation testing, in: 25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3–6, 2014, 2014, pp. 277–287, https://doi.org/10.1109/ISSRE.2014.27.

[119] J. Zhang, Scalability studies on selective mutation testing, in: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16–24, 2015, vol. 2, 2015, pp. 851–854, https://doi.org/10.1109/ICSE.2015.276.

[120] L. Zhang, M. Gligoric, D. Marinov, S. Khurshid, Operator-based and random mutant selection: better together, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11–15, 2013, 2013, pp. 92–102, https://doi.org/10.1109/ASE.2013.6693070.

[121] R. Gopinath, M.A. Alipour, I. Ahmed, C. Jensen, A. Groce, On the limits of mutation reduction strategies, in: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016, 2016, pp. 511–522, https://doi.org/10.1145/2884781.2884787.

[122] B. Kurtz, P. Ammann, J. Offutt, M.E. Delamaro, M. Kurtz, N. Gökçe, Analyzing the validity of selective mutation with dominator mutants, in: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13–18, 2016, 2016, pp. 571–582, https://doi.org/10.1145/2950290.2950322.

[123] M. Gligoric, L. Zhang, C. Pereira, G. Pokam, Selective mutation testing for concurrent code, in: International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15–20, 2013, 2013, pp. 224–234, https://doi.org/10.1145/2483760.2483773.

[124] G.K. Kaminski, P. Ammann, Using a fault hierarchy to improve the efficiency of DNF logic mutation testing, in: Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, CO, USA, April 1–4, 2009, 2009, pp. 386–395, https://doi.org/10.1109/ICST.2009.13.

[125] M. Papadakis, Y.L. Traon, Effective fault localization via mutation analysis: a selective mutation approach, in: Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea, March 24–28, 2014, 2014, pp. 1293–1300, https://doi.org/10.1145/2554850.2554978.

[126] M. Polo, M. Piattini, I.G.R. de Guzmán, Decreasing the cost of mutation testing with second-order mutants, Softw. Test. Verif. Reliab. 19 (2) (2009) 111–131, https://doi.org/10.1002/stvr.392.

[127] M. Papadakis, N. Malevris, M. Kintis, Mutation testing strategies—a collateral approach, in: ICSOFT 2010—Proceedings of the Fifth International Conference on Software and Data Technologies, vol, 2, Athens, Greece, July 22–24, 2010, 2010, pp. 325–328.

[128] P.R. Mateo, M.P. Usaola, J.L.F. Alemán, Validating second-order mutation at system level, IEEE Trans. Softw. Eng. 39 (4) (2013) 570–587, https://doi.org/10.1109/TSE.2012.39.

[129] A. Parsai, A. Murgia, S. Demeyer, A model to estimate first-order mutation coverage from higher-order mutation coverage, in: 2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016, Vienna, Austria, August 1–3, 2016, 2016, pp. 365–373, https://doi.org/10.1109/QRS.2016.48.

[130] R. Just, B. Kurtz, P. Ammann, Inferring mutant utility from program context, in: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10–14, 2017, 2017, pp. 284–294, https://doi.org/10.1145/3092703.3092732.

[131] C. Sun, F. Xue, H. Liu, X. Zhang, A path-aware approach to mutant reduction in mutation testing, Inf. Softw. Technol. 81 (2017) 65–81, https://doi.org/10.1016/j.infsof.2016.02.006.

[132] D. Gong, G. Zhang, X. Yao, F. Meng, Mutant reduction based on dominance relation for weak mutation testing, Inf. Softw. Technol. 81 (2017) 82–96, https://doi.org/10.1016/j.infsof.2016.05.001.

[133] C. Iida, S. Takada, Reducing mutants with mutant killable precondition, in: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2017, pp. 128–133, https://doi.org/10.1109/ICSTW.2017.29.

[134] M. Patrick, M. Oriol, J.A. Clark, MESSI: mutant evaluation by static semantic interpretation, in: Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17–21, 2012, 2012, pp. 711–719, https://doi.org/10.1109/ICST.2012.161.

[135] M. Patrick, R. Alexander, M. Oriol, J.A. Clark, Probability-based semantic interpretation of mutants, in: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31–April 4, 2014, Cleveland, OH, USA, 2014, pp. 186–195, https://doi.org/10.1109/ICSTW.2014.18.

[136] M. Sridharan, A.S. Namin, Prioritizing mutation operators based on importance sampling, in: IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, November 1–4, 2010, 2010, pp. 378–387, https://doi.org/10.1109/ISSRE.2010.16.

[137] A.S. Namin, X. Xue, O. Rosas, P. Sharma, MuRanker: a mutant ranking tool, Softw. Test. Verif. Reliab. 25 (5–7) (2015) 572–604, https://doi.org/10.1002/stvr.1542.

[138] J. Nam, D. Schuler, A. Zeller, Calibrated mutation testing, in: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, March 21–25, 2011, Workshop Proceedings, 2011, pp. 376–381, https://doi.org/10.1109/ICSTW.2011.57.

[139] L. Inozemtseva, H. Hemmati, R. Holmes, Using fault history to improve mutation reduction, in: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18–26, 2013, 2013, pp. 639–642, https://doi.org/10.1145/2491411.2494586.

[140] R.H. Untch, A.J. Offutt, M.J. Harrold, Mutation analysis using mutant schemata, in: Proceedings of the 1993 International Symposium on Software Testing and Analysis, ISSTA 1993, Cambridge, MA, USA, June 28–30, 1993, 1993, pp. 139–148, https://doi.org/10.1145/154183.154265.

[141] M. Papadakis, 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, November 1–4, 2010, 2010, pp. 121–130, https://doi.org/10.1109/ISSRE.2010.38.

[142] B. Wang, Y. Xiong, Y. Shi, L. Zhang, D. Hao, Faster mutation analysis via equivalence modulo states, in: ISSTA 2017 Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017, pp. 295–306, https://doi.org/10.1145/3092703.3092714.

[143] S. Tokumoto, H. Yoshida, K. Sakamoto, S. Honiden, MuVM: higher order mutation analysis virtual machine for C, in: 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11–15, 2016, 2016, pp. 320–329, https://doi.org/10.1109/ICST.2016.18.

[144] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y.L. Traon, 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, https://doi.org/10.1109/ICST.2015.7102607.

[145] M. Marcozzi, S. Bardin, N. Kosmatov, M. Papadakis, V. Prevosto, L. Correnson, Time to clean your test objectives, in: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27–3 June, 2018, 2018,

[146] M. Marcozzi, M. Delahaye, S. Bardin, N. Kosmatov, V. Prevosto, Generic and effective specification of structural test objectives, in: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2017, pp. 436–441, https://doi.org/10.1109/ICST.2017.48.

[147] H. Coles, T. Laurent, C. Henard, M. Papadakis, A. Ventresque, PIT: a practical mutation testing tool for Java (demo), in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18–20, 2016, 2016, pp. 449–452, https://doi.org/10.1145/2931037.2948707.

[148] F. Hariri, A. Shi, H. Converse, S. Khurshid, D. Marinov, Evaluating the effects of compiler optimizations on mutation testing at the compiler IR level, in: 27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23–27, 2016, 2016, pp. 105–115, https://doi.org/10.1109/ISSRE.2016.51.

[149] M. Papadakis, N. Malevris, Mutation based test case generation via a path selection strategy, Inf. Softw. Technol. 54 (9) (2012) 915–932, https://doi.org/10.1016/j.infsof.2012.02.004.

[150] V.H.S. Durelli, J. Offutt, M.E. Delamaro, Toward harnessing high-level language virtual machines for further speeding up weak mutation testing, in: Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17–21, 2012, 2012, pp. 681–690, https://doi.org/10.1109/ICST.2012.158.

[151] X. Devroey, G. Perrouin, M. Cordy, M. Papadakis, A. Legay, P. Schobbens, A variability perspective of mutation analysis, 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. 841–844, https://doi.org/10.1145/2635868.2666610.

[152] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y.L. Traon, M. Harman, Detecting trivial mutant equivalences via compiler optimisations, IEEE Trans. Softw. Eng. PP (99) (2017) 1. ISSN: 0098-5589, https://doi.org/10.1109/TSE.2017.2684805.

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

[154] M. Kintis, N. Malevris, Using data flow patterns for equivalent mutant detection, in: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31–April 4, 2014, Cleveland, OH, USA, 2014, pp. 196–205, https://doi.org/10.1109/ICSTW.2014.21.

[155] M. Kintis, N. Malevris, MEDIC: A static analysis framework for equivalent mutant identification, Inf. Softw. Technol. 68 (2015) 1–17, https://doi.org/10.1016/j.infsof.2015.07.009.

[156] D. Holling, S. Banescu, M. Probst, A. Petrovska, A. Pretschner, Nequivack: assessing mutation score confidence, in: Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11–15, 2016, 2016, pp. 152–161, https://doi.org/10.1109/ICSTW.2016.29.

[157] M. Kintis, N. Malevris, Identifying more equivalent mutants via code similarity, in: 20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2–5, 2013, vol. 1, 2013, pp. 180–188, https://doi.org/10.1109/APSEC.2013.34.
[158] K.A. Foster, Error sensitive test cases analysis (ESTCA), IEEE Trans. Softw. Eng. 6 (3) (1980) 258–264, https://doi.org/10.1109/TSE.1980.234487.
[159] W.E. Howden, Weak mutation testing and completeness of test sets, IEEE Trans. Softw. Eng. 8 (4) (1982) 371–379, https://doi.org/10.1109/TSE.1982.235571.
[160] K. Tai, Predicate-based test generation for computer programs, in: Proceedings of the 15th International Conference on Software Engineering, Baltimore, MD, USA, May 17–21, 1993, 1993, pp, 267–276. http://portal.acm.org/citation.cfm?id=257572.257631.
[161] K. Tai, Theory of fault-based predicate testing for computer programs, IEEE Trans. Softw. Eng. 22 (8) (1996) 552–562, https://doi.org/10.1109/32.536956.
[162] M. Papadakis, Error detection methods in Java programs using the mutation method, (Masters thesis), Athens University of Economics and Business 2005.
[163] G. Kaminski, P. Ammann, J. Offutt, 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, https://doi.org/10.1145/1982595.1982608.
[164] R. Just, G.M. Kapfhammer, 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, https://doi.org/10.1109/ICST.2012.162.
[165] R. Just, F. Schweiggert, Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators, Softw. Test. Verif. Reliab. 25 (5–7) (2015) 490–507, https://doi.org/10.1002/stvr.1561.
[166] L. Fernandes, M. Ribeiro, L. Carvalho, R. Gheyi, M. Mongiovi, A. Santos, A. Cavalcanti, F. Ferrari, J.C. Maldonado, Avoiding useless mutants, in: Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, ACM, New York, NY, USA, 2017, ISBN: 978-1-4503-5524-7, pp. 187–198, https://doi.org/10.1145/3136040.3136053.
[167] T. Laurent, A. Ventresque, M. Papadakis, C. Henard, Y.L. Traon, Assessing and improving the mutation testing practice of PIT, CoRR (2016). http://arxiv.org/abs/1601.02351.
[168] B. Lindström, A. Marki, On strong mutation and subsuming mutants, in: Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11–15, 2016, 2016, pp. 112–121, https://doi.org/10.1109/ICSTW.2016.28.
[169] B. Kurtz, P. Ammann, 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, https://doi.org/10.1109/ICSTW.2015.7107454.
[170] R.A. DeMillo, A.J. Offutt, Constraint-based automatic test data generation, IEEE Trans. Softw. Eng. 17 (9) (1991) 900–910, https://doi.org/10.1109/32.92910.
[171] G. Fraser, A. Zeller, Mutation-driven generation of unit tests and oracles, IEEE Trans. Softw. Eng. 38 (2) (2012) 278–292, https://doi.org/10.1109/TSE.2011.93.
[172] M. Harman, Y. Jia, W.B. Langdon, Strong higher order mutation-based test data generation, in: SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5–9, 2011, 2011, pp. 212–222, https://doi.org/10.1145/2025113.2025144.

[173] S. Anand, E.K. Burke, T.Y. Chen, J.A. Clark, M.B. Cohen, W. Grieskamp, M. Harman, M.J. Harrold, P. McMinn, An orchestrated survey of methodologies for automated software test case generation, J. Syst. Softw. 86 (8) (2013) 1978–2001, https://doi.org/10.1016/j.jss.2013.02.061.
[174] F. Wotawa, M. Nica, B.K. Aichernig, Generating distinguishing tests using the Minion constraint solver, in: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7–9, 2010, Workshops Proceedings, 2010, pp. 325–330, https://doi.org/10.1109/ICSTW.2010.11.
[175] S. Nica, On the improvement of the mutation score using distinguishing test cases, in: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21–25, 2011, 2011, pp. 423–426, https://doi.org/10.1109/ICST.2011.40.
[176] M. Papadakis, N. Malevris, An effective path selection strategy for mutation testing, in: 16th Asia-Pacific Software Engineering Conference, APSEC 2009, December 1–3, 2009, Batu Ferringhi, Penang, Malaysia, 2009, pp. 422–429, https://doi.org/10.1109/APSEC.2009.68.
[177] M. Papadakis, N. Malevris, Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing, Softw. Q. J. 19 (4) (2011) 691–723, https://doi.org/10.1007/s11219-011-9142-y.
[178] S. Anand, C.S. Pasareanu, W. Visser, JPF-SE: a symbolic execution extension to Java PathFinder, in: Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24–April 1, 2007, 2007, pp. 134–138, https://doi.org/10.1007/978-3-540-71209-1_12.
[179] C. Cadar, D. Dunbar, D.R. Engler, KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs, in: 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8–10, 2008, San Diego, CA, USA, Proceedings, 2008, pp, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.
[180] H. Riener, R. Bloem, G. Fey, Test case generation from mutants using model checking techniques, in: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, March 21–25, 2011, Workshop Proceedings, 2011, pp. 388–397, https://doi.org/10.1109/ICSTW.2011.55.
[181] M. Papadakis, N. Malevris, M. Kallia, Towards automating the generation of mutation tests, in: The 5th Workshop on Automation of Software Test, AST 2010, May 3–4, 2010, Cape Town, South Africa, 2010, pp. 111–118, https://doi.org/10.1145/1808266.1808283.
[182] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, H. Mei, Test generation via dynamic symbolic execution for mutation testing, in: 26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12–18, 2010, Timisoara, Romania, 2010, pp. 1–10, https://doi.org/10.1109/ICSM.2010.5609672.
[183] S. Bardin, N. Kosmatov, F. Cheynier, Efficient leveraging of symbolic execution to advanced coverage criteria, in: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014–April 4, 2014, Cleveland, OH, USA, 2014, pp. 173–182, https://doi.org/10.1109/ICST.2014.30.
[184] K. Jamrozik, G. Fraser, N. Tillmann, J. de Halleux, Augmented dynamic symbolic execution, in: IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3–7, 2012, 2012, pp. 254–257, https://doi.org/10.1145/2351676.2351716.
[185] F.C.M. Souza, M. Papadakis, Y.L. Traon, M.E. Delamaro, Strong mutation-based test data generation using hill climbing, in: Proceedings of the 9th International Workshop on Search-Based Software Testing, SBST@ICSE 2016, Austin, TX, USA, May 14–22, 2016, 2016, pp. 45–54, https://doi.org/10.1145/2897010.2897012.

[186] K. Ayari, S. Bouktif, G. Antoniol, Automatic mutation test input data generation via ant colony, in: Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7–11, 2007, 2007, pp. 1074–1081, https://doi.org/10.1145/1276958.1277172.

[187] G. Fraser, A. Zeller, Mutation-driven generation of unit tests and oracles, in: Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12–16, 2010, 2010, pp. 147–158, https://doi.org/10.1145/1831708.1831728.

[188] M. Papadakis, N. Malevris, Automatic mutation based test data generation, in: 13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Companion Material Proceedings, Dublin, Ireland, July 12–16, 2011, 2011, pp. 247–248, https://doi.org/10.1145/2001858.2001997.

[189] M. Papadakis, N. Malevris, Killing mutants effectively a search based approach, in: Knowledge-Based Software Engineering—Proceedings of the Tenth Conference on Knowledge-Based Software Engineering, JCKBSE 2012, Rodos, Greece, August 23–26, 2012, 2012, pp. 217–226, https://doi.org/10.3233/978-1-61499-094-9-217.

[190] M. Papadakis, N. Malevris, Searching and generating test inputs for mutation testing, SpringerPlus 2 (1) (2013) 121. ISSN: 2193-1801, https://doi.org/10.1186/2193-1801-2-121.

[191] M. Patrick, R. Alexander, M. Oriol, J.A. Clark, Using mutation analysis to evolve subdomains for random testing, in: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18–22, 2013, 2013, pp. 53–62, https://doi.org/10.1109/ICSTW.2013.14.

[192] G. Fraser, A. Arcuri, Achieving scalable mutation-based generation of whole test suites, Empir. Softw. Eng. 20 (3) (2015) 783–812, https://doi.org/10.1007/s10664-013-9299-z.

[193] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, 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, https://doi.org/10.1145/2635868.2635921.

[194] Y. Lou, D. Hao, L. Zhang, Mutation-based test-case prioritization in software evolution, in: 26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2–5, 2015, 2015, pp. 46–57, https://doi.org/10.1109/ISSRE.2015.7381798.

[195] L. Zhang, D. Marinov, S. Khurshid, Faster mutation testing inspired by test prioritization and reduction, in: International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15–20, 2013, 2013, pp. 235–245, https://doi.org/10.1145/2483760.2483782.

[196] R. Just, G.M. Kapfhammer, F. Schweiggert, Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis, in: 23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27–30, 2012, 2012, pp. 11–20, https://doi.org/10.1109/ISSRE.2012.31.

[197] Q. Zhu, A. Panichella, A. Zaidman, Speeding-up mutation testing via data compression and state infection, in: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2017, pp. 103–109, https://doi.org/10.1109/ICSTW.2017.25.

[198] M.E. Delamaro, Proteum - a mutation analysis based testing environmen, (Masters thesis), University of São Paulo, Sao Paulo, Brazil, 1993.

[199] S. Kim, Y. Ma, Y.R. Kwon, Combining weak and strong mutation for a non-interpretive Java mutation system, Softw. Test. Verif. Reliab. 23 (8) (2013) 647–668, https://doi.org/10.1002/stvr.1480.

[200] R. Just, M.D. Ernst, G. Fraser, Efficient mutation analysis by propagating and partitioning infected execution states, in: International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA, July 21–26, 2014, 2014, pp. 315–326, https://doi.org/10.1145/2610384.2610388.
[201] P.R. Mateo, M.P. Usaola, Mutant execution cost reduction: through MUSIC (mutant schema improved with extra code), in: Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17–21, 2012, 2012, pp. 664–672, https://doi.org/10.1109/ICST.2012.156.
[202] P.R. Mateo, M.P. Usaola, Reducing mutation costs through uncovered mutants, Softw. Test. Verif. Reliab. 25 (5–7) (2015) 464–489, https://doi.org/10.1002/stvr.1534.
[203] K.N. King, A.J. Offutt, A Fortran language system for mutation-based software testing, Softw. Pract. Exper. 21 (7) (1991) 685–718, https://doi.org/10.1002/spe.4380210704.
[204] M. Papadakis, M.E. Delamaro, Y.L. Traon, Proteum/FL: a tool for localizing faults using mutation analysis, in: 13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2013, Eindhoven, Netherlands, September 22–23, 2013, 2013, pp. 94–99, https://doi.org/10.1109/SCAM.2013.6648189.
[205] D. Schuler, 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, https://doi.org/10.1145/1595696.1595750.
[206] P.R. Mateo, M.P. Usaola, Parallel mutation testing, Softw. Test. Verif. Reliab. 23 (4) (2013) 315–350, https://doi.org/10.1002/stvr.1471.
[207] M. Gligoric, V. Jagannath, Q. Luo, D. Marinov, Efficient mutation testing of multithreaded code, Softw. Test. Verif. Reliab. 23 (5) (2013) 375–403, https://doi.org/10.1002/stvr.1469.
[208] M. Gligoric, V. Jagannath, D. Marinov, MuTMuT: efficient exploration for mutation testing of multithreaded code, in: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7–9, 2010, 2010, pp. 55–64, https://doi.org/10.1109/ICST.2010.33.
[209] P. Gong, R. Zhao, Z. Li, Faster mutation-based fault localization with a novel mutation execution strategy, in: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13–17, 2015, 2015, pp. 1–10, https://doi.org/10.1109/ICSTW.2015.7107448.
[210] L. Zhang, D. Marinov, L. Zhang, 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, https://doi.org/10.1145/2338965.2336793.
[211] C.J. Wright, G.M. Kapfhammer, P. McMinn, Efficient mutation analysis of relational database structure using mutant schemata and parallelisation, in: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18–22, 2013, 2013, pp. 63–72, https://doi.org/10.1109/ICSTW.2013.15.
[212] C. Zhou, P.G. Frankl, Inferential checking for mutants modifying satabase states, in: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21–25, 2011, 2011, pp. 259–268, https://doi.org/10.1109/ICST.2011.63.
[213] M. Patrick, A.P. Craig, N.J. Cunniffe, M. Parry, C.A. Gilligan, Testing stochastic software using pseudo-oracles, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18–20, 2016, 2016, pp. 235–246, https://doi.org/10.1145/2931037.2931063.

[214] M.J. Rutherford, A. Carzaniga, A.L. Wolf, Evaluating test suites and adequacy criteria using simulation-based models of distributed systems, IEEE Trans. Softw. Eng. 34 (4) (2008) 452–470, https://doi.org/10.1109/TSE.2008.33.

[215] B.J.M. Grün, D. Schuler, A. Zeller, The impact of equivalent mutants, in: Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, CO, USA, April 1–4, 2009, Workshops Proceedings, 2009, pp. 192–199, https://doi.org/10.1109/ICSTW.2009.37.

[216] D. Schuler, V. Dallmeier, A. Zeller, Efficient mutation testing by checking invariant violations, in: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19–23, 2009, 2009, pp. 69–80, https://doi.org/10.1145/1572272.1572282.

[217] D. Schuler, A. Zeller, (Un-)covering equivalent mutants, in: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7–9, 2010, 2010, pp. 45–54, https://doi.org/10.1109/ICST.2010.30.

[218] D. Schuler, A. Zeller, Covering and uncovering equivalent mutants, Softw. Test. Verif. Reliab. 23 (5) (2013) 353–374, https://doi.org/10.1002/stvr.1473.

[219] B. Schwarz, D. Schuler, A. Zeller, Breeding high-impact mutations, in: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, March 21–25, 2011, Workshop Proceedings, 2011, pp. 382–387, https://doi.org/10.1109/ICSTW.2011.56.

[220] M. Papadakis, Y.L. Traon, Mutation testing strategies using mutant classification, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18–22, 2013, 2013, pp. 1223–1229, https://doi.org/10.1145/2480362.2480592.

[221] M. Papadakis, M.E. Delamaro, Y.L. Traon, Mitigating the effects of equivalent mutants with mutant classification strategies, Sci. Comput. Program. 95 (2014) 298–319, https://doi.org/10.1016/j.scico.2014.05.012.

[222] M. Kintis, M. Papadakis, N. Malevris, Isolating first order equivalent mutants via second order mutation, in: Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17–21, 2012, 2012, pp. 701–710, https://doi.org/10.1109/ICST.2012.160.

[223] M. Kintis, M. Papadakis, N. Malevris, Employing second-order mutation for isolating first-order equivalent mutants, Softw. Test. Verif. Reliab. 25 (5–7) (2015) 508–535, https://doi.org/10.1002/stvr.1529.

[224] M.P. Usaola, P.R. Mateo, B.P. Lamancha, Reduction of test suites using mutation, in: Fundamental Approaches to Software Engineering–15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24–April 1, 2012, 2012, pp. 425–438, https://doi.org/10.1007/978-3-642-28872-2_29.

[225] D. Hao, L. Zhang, X. Wu, H. Mei, G. Rothermel, On-demand test suite reduction, in: 34th International Conference on Software Engineering, ICSE 2012, June 2–9, 2012, Zurich, Switzerland, 2012, pp. 738–748, https://doi.org/10.1109/ICSE.2012.6227144.

[226] M.A. Alipour, A. Shi, R. Gopinath, D. Marinov, A. Groce, Evaluating non-adequate test-case reduction, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3–7, 2016, 2016, pp. 16–26, https://doi.org/10.1145/2970276.2970361.

[227] D.C. Nguyen, A. Marchetto, P. Tonella, Change sensitivity based prioritization for audit testing of webservice compositions, in: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, March 21–25, 2011, Workshop Proceedings, 2011, pp. 357–365, https://doi.org/10.1109/ICSTW.2011.50.

[228] D. Tengeri, L. Vidács, Á. Beszédes, J. Jász, G. Balogh, B. Vancsics, T. Gyimóthy, Relating code coverage, mutation score and test suite reducibility to defect density, in: Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11–15, 2016, 2016, pp. 174–179, https://doi.org/10.1109/ICSTW.2016.25.

[229] B. Kurtz, P. Ammann, J. Offutt, M. Kurtz, Are we there yet? How redundant and equivalent mutants affect determination of test completeness, in: Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11–15, 2016, 2016, pp. 142–151, https://doi.org/10.1109/ICSTW.2016.41.

[230] G. Fraser, A. Zeller, Generating parameterized unit tests, in: Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17–21, 2011, 2011, pp. 364–374, https://doi.org/10.1145/2001420.2001464.

[231] T. Knauth, C. Fetzer, P. Felber, Assertion-driven development: assessing the quality of contracts using meta-mutations, in: Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, CO, USA, April 1–4, 2009, Workshops Proceedings, 2009, pp. 182–191, https://doi.org/10.1109/ICSTW.2009.40.

[232] G. Fraser, A. Arcuri, EvoSuite: automatic test suite generation for object-oriented software, in: SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5–9, 2011, 2011, pp. 416–419, https://doi.org/10.1145/2025113.2025179.

[233] H. Yoshida, S. Tokumoto, M.R. Prasad, I. Ghosh, T. Uehara, FSX: fine-grained incremental unit test generation for C/C++ programs, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18–20, 2016, 2016, pp. 106–117, https://doi.org/10.1145/2931037.2931055.

[234] H. Yoshida, S. Tokumoto, M.R. Prasad, I. Ghosh, T. Uehara, FSX: a tool for fine-grained incremental unit test generation for C/C++ programs, in: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13–18, 2016, 2016, pp. 1052–1056, https://doi.org/10.1145/2950290.2983937.

[235] S. Mirshokraie, A. Mesbah, K. Pattabiraman, PYTHIA: generating test cases with oracles for JavaScript applications, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11–15, 2013, 2013, pp. 610–615, https://doi.org/10.1109/ASE.2013.6693121.

[236] S. Mirshokraie, A. Mesbah, K. Pattabiraman, JSEFT: automated javascript unit test generation, in: 8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13–17, 2015, 2015, pp. 1–10, https://doi.org/10.1109/ICST.2015.7102595.

[237] M. Staats, G. Gay, M.P.E. Heimdahl, Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing, in: 34th International Conference on Software Engineering, ICSE 2012, June 2–9, 2012, Zurich, Switzerland, 2012, pp. 870–880, https://doi.org/10.1109/ICSE.2012.6227132.

[238] G. Gay, M. Staats, M.W. Whalen, M.P.E. Heimdahl, Automated oracle data selection support, IEEE Trans. Softw. Eng. 41 (11) (2015) 1119–1137, https://doi.org/10.1109/TSE.2015.2436920.

[239] G. Jahangirova, D. Clark, M. Harman, P. Tonella, Test oracle assessment and improvement, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18–20, 2016, 2016, pp. 247–258, https://doi.org/10.1145/2931037.2931062.

[240] E.T. Barr, M. Harman, P. McMinn, M. Shahbaz, S. Yoo, The oracle problem in software testing: a survey, IEEE Trans. Softw. Eng. 41 (5) (2015) 507–525, https://doi.org/10.1109/TSE.2014.2372785.
[241] M. Papadakis, Y.L. Traon, Using mutants to locate "unknown" faults, in: Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17–21, 2012, 2012, pp. 691–700, https://doi.org/10.1109/ICST.2012.159.
[242] T.T. Chekam, M. Papadakis, Y.L. Traon, Assessing and comparing mutation-based fault localization techniques, CoRR (2016) http://arxiv,org/abs/1607.05512.
[243] S. Moon, Y. Kim, M. Kim, S. Yoo, Ask the mutants: mutating faulty programs for fault localization, in: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014–April 4, 2014, Cleveland, OH, USA, 2014, pp. 153–162, https://doi.org/10.1109/ICST.2014.28.
[244] L. Zhang, L. Zhang, S. Khurshid, Injecting mechanical faults to localize developer faults for evolving software, in: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, Part of SPLASH 2013, Indianapolis, IN, USA, October 26–31, 2013, 2013, pp. 765–784, https://doi.org/10.1145/2509136.2509551.
[245] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, M. Kim, Mutation-based fault localization for real-world multilingual programs (T), in: 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9–13, 2015, 2015, pp. 464–475, https://doi.org/10.1109/ASE.2015.14.
[246] S. Hong, T. Kwak, B. Lee, Y. Jeon, B. Ko, Y. Kim, M. Kim, MUSEUM: debugging real-world multilingual programs using mutation analysis, Inf. Softw. Technol. 82 (2017) 80–95, https://doi.org/10.1016/j.infsof.2016.10.002.
[247] S.S. Murtaza, N.H. Madhavji, M. Gittens, Z. Li, Diagnosing new faults using mutants and prior faults, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21–28, 2011, 2011, pp. 960–963, https://doi.org/10.1145/1985793.1985959.
[248] S.S. Murtaza, A. Hamou-Lhadj, N.H. Madhavji, M. Gittens, An empirical study on the use of mutant traces for diagnosis of faults in deployed systems, J. Syst. Softw. 90 (2014) 29–44, https://doi.org/10.1016/j.jss.2013.11.1094.
[249] V. Musco, M. Monperrus, P. Preux, Mutation-based graph inference for fault localization, in: 16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2–3, 2016, 2016, pp. 97–106, https://doi.org/10.1109/SCAM.2016.24.
[250] V. Debroy, W.E. Wong, Using mutation to automatically suggest fixes for faulty programs, in: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7–9, 2010, 2010, pp. 65–74, https://doi.org/10.1109/ICST.2010.66.
[251] V. Debroy, W.E. Wong, Combining mutation and fault localization for automated program debugging, J. Syst. Softw. 90 (2014) 45–60, https://doi.org/10.1016/j.jss.2013.10.042.
[252] W. Weimer, T. Nguyen, C. Le Goues, S. Forrest, Automatically finding patches using genetic programming, in: 31st International Conference on Software Engineering, ICSE 2009, May 16–24, 2009, Vancouver, Canada, Proceedings, 2009, pp. 364–374, https://doi.org/10.1109/ICSE.2009.5070536.
[253] C. Le Goues, M. Dewey-Vogt, S. Forrest, W. Weimer, A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each, in: 34th International Conference on Software Engineering, ICSE 2012, June 2–9, 2012, Zurich, Switzerland, 2012, pp. 3–13, https://doi.org/10.1109/ICSE.2012.6227211.

[254] W. Weimer, Z.P. Fry, S. Forrest, Leveraging program equivalence for adaptive program repair: models and first results, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11–15, 2013, 2013, pp. 356–366, https://doi.org/10.1109/ASE.2013.6693094.
[255] S.H. Tan, A. Roychoudhury, relifix: automated repair of software regressions, in: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16–24, 2015, vol. 1, 2015, pp. 471–482, https://doi.org/10.1109/ICSE.2015.65.
[256] F. Long, M. Rinard, Staged program repair with condition synthesis, 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. 166–178, https://doi.org/10.1145/2786805.2786811.
[257] D. Kim, J. Nam, J. Song, S. Kim, Automatic patch generation learned from human-written patches, in: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18–26, 2013, 2013, pp. 802–811, https://doi.org/10.1109/ICSE.2013.6606626.
[258] J. Petke, S. Haraldsson, M. Harman, W. Langdon, D. White, J. Woodward, Genetic improvement of software: a comprehensive survey, IEEE Trans. Evol. Comput. PP (99) (2017) 1. ISSN: 1089-778X, https://doi.org/10.1109/TEVC.2017.2693219.
[259] S. Chandra, E. Torlak, S. Barman, R. Bodík, Angelic debugging, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21–28, 2011, 2011, pp. 121–130, https://doi.org/10.1145/1985793.1985811.
[260] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, L. Zhang, Predictive mutation testing, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18–20, 2016, 2016, pp. 342–353, https://doi.org/10.1145/2931037.2931038.
[261] J.M. Rojas, G. Fraser, Code defenders: a mutation testing game, in: Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11–15, 2016, 2016, pp. 162–167, https://doi.org/10.1109/ICSTW.2016.43.
[262] J.M. Rojas, T.D. White, B.S. Clegg, G. Fraser, Code defenders: crowdsourcing effective tests and subtle mutants with a mutation testing game, in: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017, 2017, pp, 677–688.
[263] M. Harman, Y. Jia, W.B. Langdon, A manifesto for higher order mutation testing, in: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7–9, 2010, Workshops Proceedings, 2010, pp. 80–89, https://doi.org/10.1109/ICSTW.2010.13.
[264] Y. Jia, M. Harman, Constructing subtle faults using higher order mutation testing, in: Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), September 28–29, 2008, Beijing, China, 2008, pp. 249–258, https://doi.org/10.1109/SCAM.2008.36.
[265] M. Harman, Y. Jia, P.R. Mateo, M. Polo, Angels and monsters: an empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation, in: ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden, September 15–19, 2014, 2014, pp. 397–408, https://doi.org/10.1145/2642937.2643008.
[266] E. Omar, S. Ghosh, D. Whitley, Constructing subtle higher order mutants for Java and AspectJ programs, in: IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4–7, 2013, 2013, pp. 340–349, https://doi.org/10.1109/ISSRE.2013.6698887.

[267] E. Omar, S. Ghosh, D. Whitley, Subtle higher order mutants, Inf. Softw. Technol. 81 (2017) 3–18, https://doi.org/10.1016/j.infsof.2016.01.016.
[268] E. Omar, S. Ghosh, D. Whitley, Comparing search techniques for finding subtle higher order mutants, in: Genetic and Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12–16, 2014, 2014, pp. 1271–1278, https://doi.org/10.1145/2576768.2598286.
[269] F. Wu, M. Harman, Y. Jia, J. Krinke, HOMI: searching higher order mutants for software improvement, in: Search Based Software Engineering—8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8–10, 2016, Proceedings, 2016, pp. 18–33, https://doi.org/10.1007/978-3-319-47106-8_2.
[270] D. Shin, S. Yoo, D. Bae, Diversity-aware mutation adequacy criterion for improving fault detection capability, in: Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11–15, 2016, 2016, pp. 122–131, https://doi.org/10.1109/ICSTW.2016.37.
[271] D. Shin, S. Yoo, D.H. Bae, A theoretical and empirical study of diversity-aware mutation adequacy criterion, IEEE Trans. Softw. Eng. PP (99) (2017) 1. ISSN: 0098-5589, https://doi.org/10.1109/TSE.2017.2732347. 1.
[272] C. Henard, M. Papadakis, G. Perrouin, J. Klein, 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, https://doi.org/10.1109/ICSTW.2013.30.
[273] C. Henard, M. Papadakis, Y.L. Traon, MutaLog: a tool for mutating logic formulas, in: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31–April 4, 2014, Cleveland, OH, USA, 2014, pp. 399–404, https://doi.org/10.1109/ICSTW.2014.54.
[274] C. Henard, M. Papadakis, G. Perrouin, J. Klein, Y.L. Traon, Towards automated testing and fixing of re-engineered feature models, in: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18–26, 2013, 2013, pp. 1245–1248, https://doi.org/10.1109/ICSE.2013.6606689.
[275] C. Henard, M. Papadakis, Y.L. Traon, Mutation-based generation of software product line test configurations, in: Search-Based Software Engineering—6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26–29, 2014, 2014, pp. 92–106, https://doi.org/10.1007/978-3-319-09940-8_7.
[276] R.A.M. Filho, S.R. Vergilio, A mutation and multi-objective test data generation approach for feature testing of software product lines, in: 29th Brazilian Symposium on Software Engineering, SBES 2015, Belo Horizonte, MG, Brazil, September 21–26, 2015, 2015, pp. 21–30, https://doi.org/10.1109/SBES.2015.17.
[277] R.A.M. Filho, S.R. Vergilio, A multi-objective test data generation approach for mutation testing of feature models, J. Softw. Eng. R&D 4 (2016) 4, https://doi.org/10.1186/s40411-016-0030-9.
[278] P. Arcaini, A. Gargantini, P. Vavassori, Generating tests for detecting faults in feature models, in: 8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13–17, 2015, 2015, pp. 1–10, https://doi.org/10.1109/ICST.2015.7102591.
[279] P. Arcaini, A. Gargantini, P. Vavassori, Automatic detection and removal of conformance faults in feature models, in: 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11–15, 2016, 2016, pp. 102–112, https://doi.org/10.1109/ICST.2016.10.
[280] M.B. Trakhtenbrot, Implementation-oriented mutation testing of statechart models, in: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7–9, 2010, Workshops Proceedings, 2010, pp. 120–125, https://doi.org/10.1109/ICSTW.2010.55.

[281] B.K. Aichernig, E. Jöbstl, Towards symbolic model-based mutation testing: pitfalls in expressing semantics as constraints, in: Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17–21, 2012, 2012, pp. 752–757, https://doi.org/10.1109/ICST.2012.169.

[282] B.K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, Efficient mutation killers in action, in: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21–25, 2011, 2011, pp. 120–129, https://doi.org/10.1109/ICST.2011.57.

[283] B.K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, S. Tiran, Killing strategies for model-based mutation testing, Softw. Test. Verif. Reliab. 25 (8) (2015) 716–748, https://doi.org/10.1002/stvr.1522.

[284] F. Belli, M. Beyazit, Exploiting model morphology for event-based testing, IEEE Trans. Softw. Eng. 41 (2) (2015) 113–134, https://doi.org/10.1109/TSE.2014.2360690.

[285] K. El-Fakih, A. Kolomeez, S. Prokopenko, N. Yevtushenko, Extended finite state machine based test derivation driven by user defined faults, in: First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9–11, 2008, 2008, pp. 308–317, https://doi.org/10.1109/ICST.2008.16.

[286] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, Z. Su, Guided, stochastic model-based GUI testing of Android apps, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, 2017, ISBN: 978-1-4503-5105-8, pp. 245–256, https://doi.org/10.1145/3106237.3106298.

[287] B.K. Aichernig, S. Marcovic, R. Schumi, Property-based testing with external test-case generators, in: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2017, pp. 337–346, https://doi.org/10.1109/ICSTW.2017.62.

[288] B.K. Aichernig, F. Lorber, Towards generation of adaptive test cases from partial models of determinized timed automata, in: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13–17, 2015, 2015, pp. 1–6, https://doi.org/10.1109/ICSTW.2015.7107409.

[289] B.K. Aichernig, F. Lorber, D. Nickovic, Time for mutants—model-based mutation testing with timed automata, in: Tests and Proofs—7th International Conference, TAP 2013, Budapest, Hungary, June 16–20, 2013, 2013, pp. 20–38, https://doi.org/10.1007/978-3-642-38916-0_2.

[290] K.G. Larsen, F. Lorber, B. Nielsen, U.M. Nyman, Mutation-based test-case generation with ecdar, in: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2017, pp. 319–328, https://doi.org/10.1109/ICSTW.2017.60.

[291] T. Zhou, H. Sun, J. Liu, X. Chen, D. Du, Improving testing coverage for safety-critical system by mutated specification, in: 21st Asia-Pacific Software Engineering Conference, APSEC 2014, Jeju, South Korea, December 1–4, 2014, vol, 1: Research Papers, 2014, pp. 43–46, https://doi.org/10.1109/APSEC.2014.15.

[292] S.F. Adra, P. McMinn, Mutation operators for agent-based models, in: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7–9, 2010, Workshops Proceedings, 2010, pp. 151–156, https://doi.org/10.1109/ICSTW.2010.9.

[293] M. Stephan, M.H. Alalfi, A. Stevenson, J.R. Cordy, Using mutation analysis for a model-clone detector comparison framework, in: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18–26, 2013, 2013, pp. 1261–1264, https://doi.org/10.1109/ICSE.2013.6606693.

[294] C.K. Roy, J.R. Cordy, A mutation/injection-based automatic framework for evaluating code clone detection tools, in: Second International Conference on Software

Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1–4, 2009, Workshops Proceedings, 2009, pp. 157–166, https://doi.org/10.1109/ICSTW.2009.18.

[295] M. Stephan, M.H. Alalfi, J.R. Cordy, Towards a taxonomy for simulink model mutations, in: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31–April 4, 2014, Cleveland, OH, USA, 2014, pp. 206–215, https://doi.org/10.1109/ICSTW.2014.17.

[296] I. Pill, I. Rubil, F. Wotawa, M. Nica, SIMULTATE: a toolset for fault injection and mutation testing of simulink models, in: Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11–15, 2016, 2016, pp. 168–173, https://doi.org/10.1109/ICSTW.2016.21.

[297] Y. Khan, J. Hassine, Mutation operators for the Atlas Transformation Language, in: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18–22, 2013, 2013, pp. 43–52, https://doi.org/10.1109/ICSTW.2013.13.

[298] J. Troya, A. Bergmayr, L. Burgue no, M. Wimmer, Towards systematic mutations for and with ATL model transformations, in: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13–17, 2015, 2015, pp. 1–10, https://doi.org/10.1109/ICSTW.2015.7107455.

[299] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, J. Bézivin, On the use of higher-order model transformations, in: Model Driven Architecture - Foundations and Applications: 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23–26, 2009, LNCSvol. 5562, 2009, pp. 18–33, https://doi.org/10.1007/978-3-642-02674-4_3.

[300] V. Aranega, J. Mottu, A. Etien, T. Degueule, B. Baudry, J. Dekeyser, Towards an automation of the mutation analysis dedicated to model transformation, Softw. Test. Verif. Reliab. 25 (5–7) (2015) 653–683, https://doi.org/10.1002/stvr.1532.

[301] A. Bartel, B. Baudry, F. Munoz, J. Klein, T. Mouelhi, Y.L. Traon, Model driven mutation applied to adaptative systems testing, in: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011 Workshops Proceedings, March 21–March 25, 2011, Berlin, Germany, 2011, pp. 408–413, https://doi.org/10.1109/ICSTW.2011.24.

[302] P. Arcaini, A. Gargantini, E. Riccobene, Using mutation to assess fault detection capability of model review, Softw. Test. Verif. Reliab. 25 (5–7) (2015) 629–652, https://doi.org/10.1002/stvr.1530.

[303] M. Kaplan, T. Klinger, A.M. Paradkar, A. Sinha, C. Williams, C. Yilmaz, Less is more: a minimalistic approach to UML model-based conformance test generation, in: First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9–11, 2008, 2008, pp. 82–91, https://doi.org/10.1109/ICST.2008.48.

[304] G. Fraser, F. Wotawa, Using model-checkers to generate and analyze property relevant test-cases, Softw. Q. J. 16 (2) (2008) 161–183, https://doi.org/10.1007/s11219-007-9031-6.

[305] M. Trakhtenbrot, Mutation patterns for temporal requirements of reactive systems, in: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2017, pp. 116–121, https://doi.org/10.1109/ICSTW.2017.27.

[306] A. Sullivan, K. Wang, R.N. Zaeem, S. Khurshid, Automated rest generation and mutation testing for alloy, in: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2017, pp. 264–275, https://doi.org/10.1109/ICST.2017.31.

[307] D. Xu, O. el Ariss, W. Xu, L. Wang, Testing aspect-oriented programs with finite state machines, Softw. Test. Verif. Reliab. 22 (4) (2012) 267–293, https://doi.org/10.1002/stvr.440.

[308] B. Lindström, S.F. Andler, J. Offutt, P. Pettersson, D. Sundmark, Mutating aspect-oriented models to test cross-cutting concerns, in: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13–17, 2015, 2015, pp. 1–10, https://doi.org/10.1109/ICSTW.2015.7107456.

[309] Y. Elrakaiby, T. Mouelhi, Y.L. Traon, Testing obligation policy enforcement using mutation analysis, in: Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17–21, 2012, 2012, pp. 673–680, https://doi.org/10.1109/ICST.2012.157.

[310] T. Mouelhi, F. Fleurey, B. Baudry, A generic metamodel for security policies mutation, in: First International Conference on Software Testing Verification and Validation, ICST 2008, Lillehammer, Norway, April 9–11, 2008, Workshops Proceedings, 2008, pp. 278–286, https://doi.org/10.1109/ICSTW.2008.2.

[311] A. Bertolino, S. Daoudagh, F. Lonetti, E. Marchetti, F. Martinelli, P. Mori, Testing of PolPA-based usage control systems, Softw. Q. J. 22 (2) (2014) 241–271, https://doi.org/10.1007/s11219-013-9216-0.

[312] T. Mouelhi, Y.L. Traon, B. Baudry, Transforming and selecting functional test cases for security policy testing, in: Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, CO, USA, April 1–4, 2009, 2009, pp. 171–180, https://doi.org/10.1109/ICST.2009.49.

[313] P.H. Nguyen, M. Papadakis, I. Rubab, Testing delegation policy enforcement via mutation analysis, in: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18–22, 2013, 2013, pp. 34–42, https://doi.org/10.1109/ICSTW.2013.12.

[314] J. Hwang, T. Xie, D.E. Kateb, T. Mouelhi, Y.L. Traon, Selection of regression system tests for security policy evolution, in: IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3–7, 2012, 2012, pp. 266–269, https://doi.org/10.1145/2351676.2351719.

[315] F. Dadeau, P. Héam, R. Kheddam, Mutation-based test generation from security protocols in HLPSL, in: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21–25, 2011, 2011, pp. 240–248, https://doi.org/10.1109/ICST.2011.42.

[316] F. Dadeau, P. Héam, R. Kheddam, G. Maatoug, M. Rusinowitch, Model-based mutation testing from security protocols in HLPSL, Softw. Test. Verif. Reliab. 25 (5–7) (2015) 684–711, https://doi.org/10.1002/stvr.1531.

[317] M. Papadakis, C. Henard, 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, OH, USA, 2014, pp. 1–10, https://doi.org/10.1109/ICST.2014.11.

[318] Z. Zhang, T. Wu, J. Zhang, Boundary value analysis in automatic white-box test generation, in: 26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2–5, 2015, 2015, pp. 239–249, https://doi.org/10.1109/ISSRE.2015.7381817.

[319] M. Patrick, Y. Jia, Kernel density adaptive random testing, in: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13–17, 2015, 2015, pp. 1–10, https://doi.org/10.1109/ICSTW.2015.7107451.

[320] M. Patrick, Y. Jia, KD-ART: should we intensify or diversify tests to kill mutants? Inf, Softw. Technol. 81 (2017) 36–51, https://doi.org/10.1016/j.infsof.2016.04.009.

[321] J.P. Galeotti, C.A. Furia, E. May, G. Fraser, A. Zeller, Inferring loop invariants by mutation, dynamic analysis, and static checking, IEEE Trans. Softw. Eng. 41 (10) (2015) 1019–1037, https://doi.org/10.1109/TSE.2015.2431688.

[322] C. Andrés, M.G. Merayo, M. Núñez, Passive testing of stochastic timed systems, in: Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, CO, USA, April 1–4, 2009, 2009, pp. 71–80, https://doi.org/10.1109/ICST.2009.35.

[323] C. Andrés, M.G. Merayo, C. Molinero, Advantages of mutation in passive testing: an empirical study, in: Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, CO, USA, April 1–4, 2009, Workshops Proceedings, 2009, pp. 230–239, https://doi.org/10.1109/ICSTW.2009.33.

[324] C. Andrés, M.G. Merayo, M. Núñez, Formal passive testing of timed systems: theory and tools, Softw. Test. Verif. Reliab. 22 (6) (2012) 365–405, https://doi.org/10.1002/stvr.1464.

[325] T. Pankumhang, M. Rutherford, Iterative instrumentation for code coverage in time-sensitive systems, in: 8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13–17, 2015, 2015, pp. 1–10, https://doi.org/10.1109/ICST.2015.7102594.

[326] A. Groce, I. Ahmed, C. Jensen, P.E. McKenney, How verified is my code? Falsification-driven verification (T), in: 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9–13, 2015, 2015, pp. 737–748, https://doi.org/10.1109/ASE.2015.40.

[327] J. Svajlenko, C.K. Roy, S. Duszynski, ForkSim: generating software forks for evaluating cross-project similarity analysis tools, in: 13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2013, Eindhoven, Netherlands, September 22–23, 2013, 2013, pp. 37–42, https://doi.org/10.1109/SCAM.2013.6648182.

[328] D. Bowes, T. Hall, M. Harman, Y. Jia, F. Sarro, F. Wu, Mutation-aware fault prediction, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18–20, 2016, 2016, pp. 330–341, https://doi.org/10.1145/2931037.2931039.

[329] J. Zhang, Y. Lou, L. Zhang, D. Hao, L. Zhang, H. Mei, Isomorphic regression testing: executing uncovered branches without test augmentation, in: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13–18, 2016, 2016, pp. 883–894, https://doi.org/10.1145/2950290.2950313.

[330] D.D. Nardo, F. Pastore, L.C. Briand, Generating complex and faulty test data through model-based mutation analysis, in: 8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13–17, 2015, 2015, pp. 1–10, https://doi.org/10.1109/ICST.2015.7102589.

[331] P. Arcaini, A. Gargantini, E. Riccobene, P. Vavassori, Rehabilitating equivalent mutants as static anomaly detectors in software artifacts, in: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13–17, 2015, 2015, pp. 1–6, https://doi.org/10.1109/ICSTW.2015.7107452.

[332] P. Arcaini, A. Gargantini, E. Riccobene, P. Vavassori, A novel use of equivalent mutants for static anomaly detection in software artifacts, Inf. Softw. Technol. 81 (2017) 52–64, https://doi.org/10.1016/j.infsof.2016.01.019.

[333] B. Baudry, S. Allier, M. Monperrus, Tailored source code transformations to synthesize computationally diverse program variants, in: International Symposium on Software Testing and Analysis, ISSTA 2014, San Jose, CA, USA, July 21–26, 2014, 2014, pp. 149–159, https://doi.org/10.1145/2610384.2610415.

[334] B. Lisper, B. Lindström, P. Potena, M. Saadatmand, M. Bohlin, Targeted mutation: efficient mutation analysis for testing non-functional properties, in: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2017, pp. 65–68, https://doi.org/10.1109/ICSTW.2017.18.
[335] A. Babu, mutatepy: A Mutation Testing Tool for C, http://members.femto-st.fr/pierre-cyrille-heam/mutatepy (accessed May 2017).
[336] I. Moore, Jester and Pester, 2001. (accessed May 2017). http://jester.sourceforge.net/.
[337] M.E. Delamaro, J.C. Maldonado, A.M.R. Vincenzi, Proteum/IM 2.0: an integrated mutation testing environment, in: Springer US, Boston, MA, 2001, ISBN: 978-1-4757-5939-6, pp. 91–101, https://doi.org/10.1007/978-1-4757-5939-6_17.
[338] J.H. Andrews, Y. Zhang, General test result checking with log file analysis, IEEE Trans. Softw. Eng. 29 (7) (2003) 634–648, https://doi.org/10.1109/TSE.2003.1214327.
[339] J.H. Andrews, L.C. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments? in: G, Roman, W.G. Griswold, B. Nuseibeh (Eds.), 27th International Conference on Software Engineering (ICSE 2005), May 15–21, 2005, St. Louis, MO, USA, ACM, 2005, pp. 402–411, https://doi.org/10.1145/1062455.1062530.
[340] Y. Ma, J. Offutt, Y.R. Kwon, MuJava: a mutation system for Java, in: L.J. Osterweil, H.D. Rombach, M.L. Soffa (Eds.), 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20–28, 2006, ACM, 2006, pp. 827–830, https://doi.org/10.1145/1134425.
[341] J. Offutt, Y. Ma, Y.R. Kwon, An experimental mutation system for Java, ACM SIGSOFT Softw. Eng. Notes 29 (5) (2004) 1–4, https://doi.org/10.1145/1022494.1022537.
[342] H. Do, G. Rothermel, On the use of mutation faults in empirical assessments of test case prioritization techniques, IEEE Trans. Softw. Eng. 32 (9) (2006) 733–752, https://doi.org/10.1109/TSE.2006.92.
[343] J. Tuya, M.J. Suarez-Cabal, C. de la Riva, SQLMutation: a tool to generate mutants of SQL database queries, in: Second Workshop on Mutation Analysis (Mutation 2006—ISSRE Workshops 2006), 2006, p. 1, https://doi.org/10.1109/MUTATION.2006.13.
[344] Jumble Testing Tool for Java, 2007. (accessed May 2017). http://jumble.sourceforge.net/.
[345] X. Feng, S. Marr, T. O'Callaghan, ESTP: an experimental software testing platform, in: Testing: Academic Industrial Conference—Practice and Research Techniques (taic part 2008), 2008, pp. 59–63, https://doi.org/10.1109/TAIC-PART.2008.8.
[346] R.P. Tan, S. Edwards, Evaluating automated unit testing in Sulu, in: First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9–11, 2008, IEEE Computer Society, 2008, pp. 62–71, https://doi.org/10.1109/ICST.2008.59.
[347] Y. Jia, M. Harman, MILU: a customizable, runtime-optimized higher order mutation testing tool for the full C language, in: Proceedings of the 3rd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART 2008), 2008, pp, 94–98. Windsor, UK.
[348] A. Rajan, M.W. Whalen, M. Staats, M.P.E. Heimdahl, Requirements coverage as an adequacy measure for conformance testing, in: S. Liu, T.S.E. Maibaum, K. Araki (Eds.), Formal Methods and Software Engineering, 10th International Conference on Formal Engineering Methods, ICFEM 2008, Kitakyushu-City, Japan, October 27–31, 2008, Proceedings, Lecture Notes in Computer Science, vol. 5256, Springer, 2008, pp. 86–104, https://doi.org/10.1007/978-3-540-88194-0_8.
[349] M. Weiglhofer, F. Wotawa, "On the fly" input output conformance verification, in: Proceedings of the IASTED International Conference on Software Engineering, ACTA Press, Anaheim, CA, USA, 2008, ISBN: 978-0-88986-716-1, pp. 286–291.
[350] C. Zhou, P.G. Frankl, Mutation testing for Java database applications, in: Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, CO, USA, April 1–4, 2009, 2009, pp. 396–405, https://doi.org/10.1109/ICST.2009.43.

[351] C. Zhou, P.G. Frankl, JDAMA: Java database application mutation analyser, Softw. Test. Verif. Reliab. 21 (3) (2011) 241–263, https://doi.org/10.1002/stvr.462.
[352] R. Delamare, B. Baudry, S. Ghosh, Y.L. Traon, A test-driven approach to developing pointcut descriptors in aspectJ, in: Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, CO, USA, April 1–4, 2009, 2009, pp. 376–385, https://doi.org/10.1109/ICST.2009.41.
[353] R. Delamare, B. Baudry, Y.L. Traon, AjMutator: a tool for the mutation analysis of aspectJ pointcut descriptors, in: Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, CO, USA, April 1–4, 2009, Workshops Proceedings, 2009, pp. 200–204, https://doi.org/10.1109/ICSTW.2009.41.
[354] J.J. Domínguez-Jiménez, A. Estero-Botaro, I. Medina-Bulo, A framework for mutant genetic generation for WS-BPEL, in: M. Nielsen, A. Kucera, P.B. Miltersen, C. Palamidessi, P. Tuma, F.D. Valencia (Eds.), SOFSEM 2009: Theory and Practice of Computer Science, 35th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlýn, Czech Republic, January 24–30, 2009, Lecture Notes in Computer Science, 5404, Springer, 2009, pp. 229–240, https://doi.org/10.1007/978-3-540-95891-8_23. vol.
[355] E.G. Aydal, R.F. Paige, M. Utting, J. Woodcock, Putting formal specifications under the magnifying glass: model-based testing for validation, in: Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, CO, USA, April 1–4, 2009, IEEE Computer Society, 2009, pp. 131–140, https://doi.org/10.1109/ICST.2009.20.
[356] K. Dobolyi, W. Weimer, Harnessing web-based application similarities to aid in regression testing, in: ISSRE 2009, 20th International Symposium on Software Reliability Engineering, Mysuru, Karnataka, India, November 16–19, 2009, IEEE Computer Society, 2009, pp. 71–80, https://doi.org/10.1109/ISSRE.2009.18.
[357] M. Ellims, D. Ince, M. Petre, The Csaw C mutation tool: initial results, in: Testing: Academic and Industrial Conference Practice and Research Techniques—MUTATION (TAICPART-MUTATION 2007), 2007, pp. 185–192, https://doi.org/10.1109/TAIC.PART.2007.28.
[358] A. Lakehal, I. Parissis, Structural coverage criteria for LUSTRE/SCADE programs, Softw. Test. Verif. Reliab. 19 (2) (2009) 133–154, https://doi.org/10.1002/stvr.394.
[359] J. Durães, H. Madeira, Definition of software fault emulation operators: a field data study, in: 2003 International Conference on Dependable Systems and Networks (DSN 2003), June 22–25, 2003, San Francisco, CA, USA, IEEE Computer Society, 2003, pp. 105–114, https://doi.org/10.1109/DSN.2003.1209922.
[360] L. Madeyski, N. Radyk, Judy—a mutation testing tool for Java, IET Softw. 4 (1) (2010) 32–42, https://doi.org/10.1049/iet-sen.2008.0038.
[361] R. Just, The major mutation framework: efficient and scalable mutation analysis for Java, in: International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA, July 21–26, 2014, 2014, pp. 433–436, https://doi.org/10.1145/2610384.2628053.
[362] P. Madiraju, A.S. Namin, Para(μ) - a partial and higher-order mutation tool with concurrency operators, in: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, March 21–25, 2011, Workshop Proceedings, 2011, pp. 351–356, https://doi.org/10.1109/ICSTW.2011.34.
[363] MuBPEL - A Mutation Testing Tool for WS-BPEL, 2011. (accessed May 2017). https://neptuno.uca.es/redmine/projects/sources-fm/wiki/MuBPEL/.
[364] K. Winbladh, A. Ranganathan, Evaluating test selection strategies for end-user specified flow-based applications, in: P. Alexander, C.S. Pasareanu, J.G. Hosking (Eds.), 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6–10, 2011, IEEE Computer Society, 2011, pp. 400–403, https://doi.org/10.1109/ASE.2011.6100083.

[365] A.A. Saifan, J. Dingel, J.S. Bradbury, E. Posse, Implementing and evaluating a runtime conformance checker for mobile agent systems, in: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21–25, 2011, IEEE Computer Society, 2011, pp. 269–278, https://doi.org/10.1109/ICST.2011.62.
[366] R. Just, G.M. Kapfhammer, F. Schweiggert, Using conditional mutation to increase the efficiency of mutation analysis, in: Proceedings of the 6th International Workshop on Automation of Software Test, AST 2011, Waikiki, Honolulu, HI, USA, May 23–24, 2011, 2011, pp. 50–56, https://doi.org/10.1145/1982595.1982606.
[367] H. Dan, R.M. Hierons, SMT-C: a semantic mutation testing tools for C, in: Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17–21, 2012, 2012, pp. 654–663, https://doi.org/10.1109/ICST.2012.155.
[368] M. Schirp, Mutation Testing for Ruby, 2012. (accessed May 2017). https://github.com/mbj/mutant.
[369] M. Kusano, C. Wang, CCmutator: a mutation generator for concurrency constructs in multithreaded C/C++ applications, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11–15, 2013, 2013, pp. 722–725, https://doi.org/10.1109/ASE.2013.6693142.
[370] J.S. Bradbury, J.R. Cordy, J. Dingel, Mutation operators for concurrent Java (J2SE 5.0), in: Second Workshop on Mutation Analysis (Mutation 2006–ISSRE Workshops 2006), 2006, p. 11, https://doi.org/10.1109/MUTATION.2006.10.
[371] G.M. Kapfhammer, P. McMinn, C.J. Wright, Search-based testing of relational schema integrity constraints across multiple database management systems, in: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18–22, 2013, IEEE Computer Society, 2013, pp. 31–40, https://doi.org/10.1109/ICST.2013.47.
[372] A. Bertolino, S. Daoudagh, F. Lonetti, E. Marchetti, XACMUT: XACML 2.0 mutants generator, in: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18–22, 2013, IEEE Computer Society, 2013, pp. 28–33, https://doi.org/10.1109/ICSTW.2013.11.
[373] C. Ye, H. Jacobsen, Whitening SOA testing via event exposure, IEEE Trans. Softw. Eng. 39 (10) (2013) 1444–1465, https://doi.org/10.1109/TSE.2013.20.
[374] A. Derezińska, K. Hałas, Analysis of Mutation Operators for the Python Language, in: Springer International Publishing, Cham, 2014, ISBN: 978-3-319-07013-1, pp. 155–164, https://doi.org/10.1007/978-3-319-07013-1_15.
[375] E. Omar, S. Ghosh, D. Whitley, HOMAJ: a tool for higher order mutation testing in aspectJ and Java, in: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31–April 4, 2014, Cleveland, OH, USA, 2014, pp. 165–170, https://doi.org/10.1109/ICSTW.2014.19.
[376] S. Mahajan, W.G.J. Halfond, Finding HTML presentation failures using image comparison techniques, in: I. Crnkovic, M. Chechik, P. Grünbacher (Eds.), ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden, September 15–19, 2014, ACM, 2014, pp. 91–96, https://doi.org/10.1145/2642937.2642966.
[377] K. El-Fakih, A. Simão, N. Jadoon, J.C. Maldonado, On studying the effectiveness of extended finite state machine based test selection criteria, in: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31–April 4, 2014, Cleveland, OH, USA, IEEE Computer Society, 2014, pp. 222–229, https://doi.org/10.1109/ICSTW.2014.25.

[378] K. Maruchi, H. Shin, M. Sakai, MC/DC-like structural coverage criteria for function block diagrams, in: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31–April 4, 2014, Cleveland, OH, USA, IEEE Computer Society, 2014, pp. 253–259, https://doi.org/10.1109/ICSTW.2014.27.

[379] T.A. Walsh, P. McMinn, G.M. Kapfhammer, Automatic detection of potential layout faults following changes to responsive web pages (N), in: M.B. Cohen, L. Grunske, M. Whalen (Eds.), 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9–13, 2015, IEEE Computer Society, 2015, pp. 709–714, https://doi.org/10.1109/ASE.2015.31.

[380] R. Abreu, B. Hofer, A. Perez, F. Wotawa, Using constraints to diagnose faulty spreadsheets, Softw. Q. J. 23 (2) (2015) 297–322, https://doi.org/10.1007/s11219-014-9236-4.

[381] F. Belli, M. Beyazit, A.T. Endo, A.P. Mathur, A. da Silva Simão, Fault domain-based testing in imperfect situations: a heuristic approach and case studies, Softw. Q. J. 23 (3) (2015) 423–452, https://doi.org/10.1007/s11219-014-9242-6.

[382] S.C.P.F. Fabbri, M.E. Delamaro, J.C. Maldonado, P.C. Masiero, Mutation analysis testing for finite state machines, in: 5th International Symposium on Software Reliability Engineering, ISSRE 1994, Monterey, CA, USA, November 6–9, 1994, IEEE, 1994, pp. 220–229, https://doi.org/10.1109/ISSRE.1994.341378.

[383] A. Simao, A. Petrenko, J.C. Maldonado, Comparing finite state machine test coverage criteria, IET Softw, 3 (2) (2009) 91–105.

[384] J. Guan, J. Offutt, A model-based testing technique for component-based real-time embedded systems, in: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13–17, 2015, IEEE Computer Society, 2015, pp. 1–10, https://doi.org/10.1109/ICSTW.2015.7107407.

[385] F.S. Ocariza Jr, K. Pattabiraman, A. Mesbah, Detecting inconsistencies in JavaScript MVC applications, in: A. Bertolino, G. Canfora, S.G. Elbaum (Eds.), 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16–24, 2015, vol. 1, IEEE Computer Society, 2015, pp. 325–335, https://doi.org/10.1109/ICSE.2015.52.

[386] W. Krenn, R. Schlick, S. Tiran, B. Aichernig, E. Jobstl, H. Brandl, MoMut::UML model-based mutation testing for UML, in: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), 2015, pp. 1–8, ISSN 2159–4848. https://doi.org/10.1109/ICST.2015.7102627.

[387] E.P. Enoiu, A. Causevic, D. Sundmark, P. Pettersson, A controlled experiment in testing of safety-critical embedded software, in: 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11–15, 2016, IEEE Computer Society, 2016, pp. 1–11, https://doi.org/10.1109/ICST.2016.15.

[388] R. Matinnejad, S. Nejati, L.C. Briand, T. Bruckmann, Automated test suite generation for time-continuous simulink models, in: L.K. Dillon, W. Visser, L. Williams (Eds.), Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016, ACM, 2016, pp. 595–606, https://doi.org/10.1145/2884781.2884797.

[389] C. Henard, M. Papadakis, M. Harman, Y. Jia, Y.L. Traon, Comparing white-box and black-box test prioritization, in: L.K. Dillon, W. Visser, L. Williams (Eds.), Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016, ACM, 2016, pp. 523–534, https://doi.org/10.1145/2884781.2884791.

[390] X. Devroey, G. Perrouin, P.Y. Schobbens, P. Heymans, Poster: VIBeS, transition system mutation made easy, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, ISSN 0270-5257, vol. 2, 2015, pp. 817–818, https://doi.org/10.1109/ICSE.2015.263.

[391] A. Parsai, A. Murgia, S. Demeyer, LittleDarwin: a feature-rich and extensible mutation testing framework for large and complex Java systems, in: Springer International Publishing, Cham, 2017, ISBN: 978-3-319-68972-2, pp. 148–163, https://doi.org/10.1007/978-3-319-68972-2_10.

[392] G. Rothermel, R.H. Untch, C. Chu, M.J. Harrold, Test case prioritization: an empirical study, in: 1999 International Conference on Software Maintenance, ICSM 1999, Oxford, England, UK, August 30–September 3, 1999, 1999, pp. 179–188, https://doi.org/10.1109/ICSM.1999.792604.

[393] A.S. Namin, 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, https://doi.org/10.1145/2001420.2001461.

[394] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, Analysing and comparing the effectiveness of mutation testing tools: a manual study, in: 16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2–3, 2016, 2016, pp. 147–156, https://doi.org/10.1109/SCAM.2016.28.

[395] R.V. Binder, Testing object-oriented systems: models, patterns, and tools, Addison-Wesley Longman Publishing Co,, Inc., Boston, MA, USA, 1999. ISBN: 0-201-80938-9.

[396] W. Visser, What makes killing a mutant hard, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3–7, 2016, 2016, pp. 39–44, https://doi.org/10.1145/2970276.2970345.

[397] P. Ammann, M.E. Delamaro, 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, OH, USA, 2014, pp. 21–30, https://doi.org/10.1109/ICST.2014.13.

[398] B. Kurtz, P. Ammann, M.E. Delamaro, J. Offutt, L. Deng, Mutant subsumption graphs, in: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31–April 4, 2014, Cleveland, OH, USA, 2014, pp. 176–185, https://doi.org/10.1109/ICSTW.2014.20.

[399] A. Arcuri, L. Briand, A practical guide for using statistical tests to assess randomized algorithms in software engineering, in: ICSE, 2011, ISBN: 978-1-4503-0445-0, pp. 1–10, https://doi.org/10.1145/1985793.1985795.

[400] M. Harman, P. McMinn, J.T. de Souza, S. Yoo, Search based software engineering: techniques, taxonomy, tutorial, in: Springer, Berlin, Heidelberg, 2012, ISBN: 978-3-642-25231-0, pp. 1–59, https://doi.org/10.1007/978-3-642-25231-0_1.

[401] M.E. Delamaro, J. Offutt, Assessing the influence of multiple test case selection on mutation experiments, in: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31–April 4, 2014, Cleveland, OH, USA, 2014, pp. 171–175, https://doi.org/10.1109/ICSTW.2014.22.

[402] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, Y. Le Traon, How effective mutation testing tools are? An empirical analysis of java mutation testing tools with manual analysis and real faults, Empir. Softw. Eng. (2017). https://doi.org/10.1007/s10664-017-9582-5 (accepted for publication).

[403] R. Gopinath, I. Ahmed, M.A. Alipour, C. Jensen, A. Groce, Does choice of mutation tool matter? Softw, Q. J. (2016) 1–50. ISSN: 1573-1367, https://doi.org/10.1007/s11219-016-9317-7.

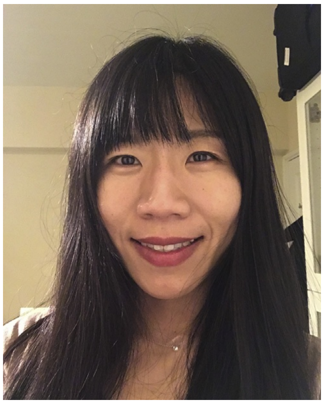
[404] A. Márki, B. Lindström, Mutation tools for Java, in: Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3–7, 2017, 2017, pp. 1364–1415, https://doi.org/10.1145/3019612.3019825.

[405] A. Arcuri, L.C. Briand, Adaptive random testing: an illusion of effectiveness? in: M,B. Dwyer, F. Tip (Eds.), Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17–21, 2011, ACM, 2011, pp. 265–275, https://doi.org/10.1145/2001420.2001452.

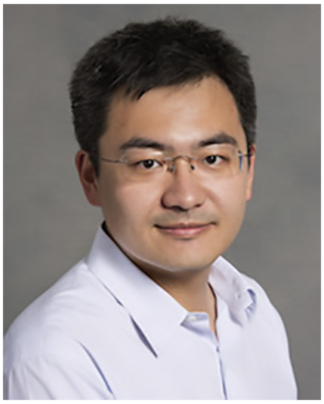
ABOUT THE AUTHORS

Mike Papadakis is a research scientist at the Interdisciplinary Centre for Security, Reliability and Trust (SnT) at the University of Luxembourg. He received a PhD diploma in Computer Science from the Athens University of Economics and Business. His research interests include software testing, static analysis, prediction modelling, mutation analysis, and search-based software engineering.

Marinos Kintis is a research associate at the Interdisciplinary Centre for Security, Reliability and Trust (SnT) at the University of Luxembourg. He received the PhD degree from the Department of Informatics of the Athens University of Economics and Business in 2016. The main topic of his dissertation was the introduction of effective techniques to ameliorate the adverse effects of the Equivalent Mutant Problem when testing software with Mutation. His main research interests include software testing, mutation testing, and program analysis. He was awarded a Best Paper Award at the 16th International Working Conference on Source Code Analysis and Manipulation (SCAM 2016) and co-organised the 13th International Workshop on Mutation Analysis (MUTATION 2018).



Jie Zhang is a final-year PhD candidate at the School of Electronics Engineering and Computer Science, Peking University, P.R. China, supervised by Lu Zhang. She is also a research associate in CREST, UCL, supervised by Earl Barr and Mark Harman. She has won the 2016 Fellowship at Microsoft Research Asia, the Top-ten Research Excellence Award of EECS, Peking University, the Lee Wai Wing Scholarship at Peking University, the 2015 National Scholarship, and so on. She served on the program committees of Mutation 2017 and Mutation 2018. Her major research interests are software testing, program analysis, end-user programming, and API mining.



Yue Jia is a lecturer in the Department of Computer Science at University College London. His research interests cover mutation testing, app store analysis, and search-based software engineering. Dr. Jia is director of MaJiCKe, an automated test data generation start up and also co-founder of Appredict, an app store analytics company, spun out from UCL's UCLappA group.

Yves Le Traon is professor at University of Luxembourg where he leads the SERVAL (SEcurity, Reasoning and VALidation) research team. His research interests within the group include (1) innovative testing and debugging techniques, (2) Android apps security and reliability using static code analysis, machine learning techniques and, (3) model-driven engineering with a focus on IoT and CPS. His reputation in the domain of software testing is acknowledged by the community. He has been General Chair of major conferences in the domain, such as the 2013 IEEE International Conference on Software Testing, Verification and Validation (ICST), and Program Chair of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS). He serves at the editorial boards of several, internationally known journals (STVR, SoSym, IEEE Transactions on Reliability) and is author of more than 140 publications in international peer-reviewed conferences and journals.

Mark Harman is currently an engineering manager at Facebook and a professor of Software Engineering in the Department of Computer Science at University College London, where he directed the CREST centre for 10 years (2006–2017) and was Head of Software Systems Engineering (2012–2017). He is widely known for work on source code analysis, software testing, app store analysis, and Search Based Software Engineering (SBSE), a field he co-founded and which has grown rapidly to include over 1600 authors spread over more than 40 countries. His SBSE and testing work has been used by many organisations including Daimler, Ericsson, Google, Huawei, Microsoft, and Visa. Prof. Harman is a co-founder (and was co-director) of Appredict, an app store analytics company, spun out from UCL's UCLappA group, and was the chief scientific advisor to Majicke, an automated test data generation start up. In February 2017, he and the other two co-founders of Majicke (Yue Jia and Ke Mao) moved to Facebook, London, in order to develop their research and technology as part of Facebook.