A Systematic Literature Review of Techniques and Metrics to Reduce the Cost of Mutation Testing

Alessandro Viola Pizzoleto[a], Fabiano Cutigi Ferrari[a,b,1,*], Jeff Offutt[b], Leo Fernandes[c], Márcio Ribeiro[d]

[a] *Computing Department, Federal University of São Carlos—São Carlos, SP, Brazil*
[b] *Department of Computer Science, George Mason University—Fairfax, VA, USA*
[c] *Informatics Coordination, Federal Institute of Alagoas—Maceió, AL, Brazil*
[d] *Computing Institute, Federal University of Alagoas—Maceió, AL, Brazil*

Abstract

Historically, researchers have proposed and applied many techniques to reduce the cost of mutation testing. It has become difficult to find all techniques and to understand the cost-benefit tradeoffs among them, which is critical to transitioning this technology to practice. This paper extends a prior workshop paper to summarize and analyze the current knowledge about reducing the cost of mutation testing through a systematic literature review. We selected 175 peer-reviewed studies, from which 153 present either original or updated contributions. Our analysis resulted in six main goals for cost reduction and 21 techniques. In the last decade, a growing number of studies explored techniques such as selective mutation, evolutionary algorithms, control-flow analysis, and higher-order mutation. Furthermore, we characterized 18 metrics, with particular interest in the number of mutants to be executed, test cases required, equivalent mutants generated and detected, and mutant execution speedup. We found that cost reduction for mutation is increasingly becoming interdisciplinary, often combining multiple techniques. Additionally, measurements vary even for studies that use the same techniques. Researchers can use our results to find more detailed information about particular techniques, and to design comparable and reproducible experiments.

Keywords: Mutation analysis, mutation testing, cost reduction, systematic review

1. Introduction

Mutation testing [2, 36, 69] (*program mutation*, *mutation analysis*, or simply *mutation*) is a testing criterion that has been extensively studied yet lightly used in the last four

*Corresponding author
Email addresses: alessandro.pizzoleto@ufscar.br (Alessandro Viola Pizzoleto), fcferrari@ufscar.br (Fabiano Cutigi Ferrari), offutt@gmu.edu (Jeff Offutt), leonardo.oliveira@ifal.edu.br (Leo Fernandes), marcio@ic.ufal.br (Márcio Ribeiro)

[1]Was a visiting researcher at George Mason University, Fairfax, VA, USA, in 2017 and 2018.

decades [54, 82, 139, 144]. Mutation creates modified versions of a program, called *mutants*, by applying *mutation operators* that make small changes to the software to either simulate faults or guide the tester to edge cases [36]. Testers are expected to find or design tests that cause these mutants to fail, that is, behave differently from the original un-mutated program. If a test case causes a mutant to fail, then that mutant is said to be *killed*; otherwise, the mutant remains alive. Mutation can be used to help testers design high quality tests or to evaluate and improve existing test sets.

Mutation testing has been empirically shown to be stronger than test criteria such as control-flow based testing and data-flow based testing [58, 110, 121, 123]. Moreover, it has been used to measure the quality of tests and other test criteria in numerous studies [11, 48, 85]. Despite its effectiveness, several factors make it expensive and therefore difficult to use in practice: large sets of mutants that must be executed, sometimes many times; generation of test cases to kill the mutants; the number of tests needed; and equivalent mutants [82, 100, 139, 164]. For even small methods with a few dozen lines of code, hundreds of mutants may be generated. Some mutants are trivially killed (that is, killed by all tests) and some are easily killed (that is, killed by many tests), yet some are more difficult and require detailed analysis by the tester. It is also very difficult (in fact undecidable in general [24]) to determine whether a mutant is equivalent or is simply difficult to kill. And not surprisingly, the most difficult-to-kill mutants often lead to valuable test cases.

These costs have been a major obstacle to practical adoption of mutation testing. For instance, some say the difficulty of identifying equivalent mutants is the primary reason mutation is not used more widely [82, 117]. Researchers have therefore invented many ways to reduce the cost, focusing on several goals. Specific goals include reducing the number of mutants [2, 122, 138], not creating certain mutants [119], speeding up mutant execution [106, 179, 187], automating test set generation [37, 38], minimizing or prioritizing test sets [167], and automatically identifying equivalent mutants [129].

Historically, many techniques to reduce mutation testing costs have been proposed, developed, and studied. The sheer volume of papers and the fact that they are published in dozens of different conferences and journals makes it hard to find them all, and even more difficult to understand the cost-benefit tradeoffs among them [65, 89, 108]. To help current and future researchers understand what has been done and discover new directions, as well as to organize the knowledge to support technology transition, we have performed a systematic literature review (SLR) to characterize the history and the state-of-the-art of mutation testing cost reduction. We previously presented preliminary results from our SLR in a workshop paper [54], with general classifications of primary studies. This paper greatly extends and updates the prior short paper, and makes four key contributions:

- It presents comprehensive classifications for cost reduction goals and cost reduction techniques, including lists of references to selected studies.
- It provides an overview of all selected studies grouped by cost reduction goal, which is complemented by extensive online material [55].

- It characterizes metrics used to measure cost reduction, including lists of references to selected studies.
- It summarizes, analyzes, and discusses the cost savings and test effectiveness as reported in the selected studies.

We analyzed 175 peer-reviewed, published, primary studies[2]. We down-selected these to 153 studies by removing studies that were either extended or updated by later studies (*subsumed*). We then characterized 6 main goals related to cost reduction, and then identified 21 cost reduction techniques. We also identified and characterized 18 metrics that have been used to measure cost reduction. In addition to the fact that cost reduction research has been increasing, we found that cost reduction techniques are becoming interdisciplinary and are more frequently combined. Moreover, we found out that cost reduction techniques have been measured in many ways, and the results vary even for studies that apply the same cost reduction technique.

The remainder of this paper is organized as follows. Section 2 provides an overview of mutation testing and the need to reduce its costs. Section 3 provides details of our SLR protocol, including goals, research questions (RQ1, RQ2 and RQ3), and the criteria and procedures we used to select and analyze the studies. Section 4 summarizes results from our search. Section 5 answers research question RQ1, by analyzing the results with a focus on the goals of the studies and the cost reduction techniques. Section 6 provides answers to research questions RQ2 and RQ3. It first lists metrics used to measure cost reduction, then analyzes the results of the selected studies regarding cost reduction measurements and achieved mutation scores. We summarize threats to the validity of this SLR in Section 7, and describe related research in Section 8. Finally, conclusions, implications, and future work are presented in Section 9.

2. Background

Recent literature reviews [54, 82, 144, 173] have shown that research in mutation testing has been steadily increasing through the last four decades. Mutation is a fault-based testing criterion that creates modified versions of the program, called *mutants* [36]. Testers use mutation to design tests and to evaluate externally created tests [11, 48, 85]. Mutation has also been used to assess other testing criteria. Papadakis et al. recently summarized 217 papers published from 2008 and 2017 that used mutation testing to perform test assessment [144].

Mutation is based on two fundamental hypotheses: the *Competent Programmer Hypothesis* and the *Coupling Effect Hypothesis* [36]. The *Competent Programmer Hypothesis* says that although programmers do not write correct programs, they write programs that are *close* to being correct. The coupling effect hypothesizes that test data that find faults that

[2]*Primary study* is a term used in evidence-based research [105] to describe research results from well-founded experimental procedures or from incipient research approaches. SLRs, on the other hand, are *secondary studies*.

are very close to the original program (that is, mutants) can also find more complex faults. That is, complex faults are *coupled* to simple faults [36]. Both hypotheses, taken together, mean that tests that kill all or most simple mutants will probably kill more complex mutants.

Mutations operators modify the program under test to create mutants. For example, an arithmetic operator would change the expression $(a + b)$ to $(a * b)$, $(a - b)$, and (a/b). Mutation operators use fault taxonomies that are usually based on studies of faults in real programs. Mutation operators are applied to a program P to create a set of mutants M. Each test t in a test set T is run against each mutant m, $m \in M$. If $m(t) \neq P(t)$ for some t, then we say that t has *killed* m. If not, the tester should find or design a test that kills m. If m and P are equivalent, then $P(t) = m(t)$ for all possible test cases.

As originally conceived [36], mutation testing was applied in four steps: (1) execution of the original program; (2) generation of mutants; (3) execution of the mutants; and (4) analysis of the mutants. After each cycle, the *mutation score* is calculated with Equation 1. In the mutation score, K is the number of mutants of P killed by T; M is the total number of mutants created from P; and E is the number of mutants of P that are equivalent to P. The mutation score is a value in the interval $[0, 1]$ that reflects the quality of the test set with respect to the mutants. The closer to 1 the mutation score is, the better the test set is.

$$MS = \frac{K}{M - E} \tag{1}$$

Steps 3 (execution) and 4 (analysis) are very costly, primarily due to two factors: the large number of executions and the need to hand-analyze live mutants. This paper characterizes research into reducing the cost of mutation testing. Step 3 has been highly automated, whereas step 4 is usually done by hand. The cost of step 4 is considered by some to be the main impediment for the practical adoption of mutation testing. Part of this analysis involves detecting equivalent mutants, which is a generally undecidable problem [24]. According to Madeyski et al. [117], several partial automated solutions to the *equivalent mutant problem* have been developed. However, these solutions are limited and thus equivalent mutants are usually found by hand.

3. Study Setup

A systematic literature review (SLR) is a rigorous approach to identify, evaluate, and interpret all available evidence about a particular topic of interest [104]. A basic requirement for SLRs is to document the process to ensure auditability and reproducibility [104].

The following subsections summarize the protocol we used. The complete protocol is provided on a companion website [55].

3.1. Goals and Research Questions

The general goal of this SLR is to summarize and analyze the history and state-of-the-art of efforts to reduce the cost of mutation testing. We define three research questions:

- (RQ1) Which techniques support cost reduction of mutation testing?

- (RQ2) Which metrics have been used to measure the cost reduction of mutation testing?
- (RQ3) What are the savings (benefit) and loss of effectiveness (as proxied by mutation score) for the techniques?

These research questions are addressed in Sections 5 (RQ1) and 6 (RQ2 and RQ3).

3.2. Control Group and Study Selection Criteria

For this SLR, the baseline dataset to assess the study selection process (the *control group* [18]) includes cost reduction-related studies described in Jia and Harman's survey [82]. We applied the following inclusion (I) and exclusion (E) selection criteria to the studies:

- (**I1**): The study was included if it proposes an approach to reduce the cost of mutation testing.
- (**I2**): The study was included if it applies an approach to reduce the cost of mutation testing.
- (**I3**): The study was included if it is a primary study, peer-reviewed, and published either in a conference or a scientific journal.
- (**E1**): The study was excluded if it mentions or uses a technique that can be used to reduce the cost of mutation testing, but cost reduction is not a main goal of that study.
- (**E2**): The study was excluded if it uses mutation testing information to assess test quality (*e.g.* the use of mutation score to assess the quality of test sets generated based on testing criteria other than mutation testing).

We selected studies that first passed I3, and then either I1 or I2 (I3 $\wedge$ (I1 $\vee$ I2)). We discarded studies that passed E1 or E2 (E1 $\vee$ E2). Notice that we use the term *approach* to refer to research that applies either classical cost reduction techniques (*e.g.* selective mutation and random mutation) or more contemporary techniques such as evolutionary algorithms and minimal mutation. As explained in Section 5.3, we found that several studies applied more than one technique in combination. Moreover, we selected studies that had a clear goal of reducing costs, either: (i) by explicitly or implicitly stating this goal in the approach description; (ii) while defining experimental goals; and/or (iii) by experimental measurements and analyses. We discarded studies that simply employed a particular mutation tool, or applied a subset of mutation operators, but did not measure cost reduction either theoretically or empirically.

3.3. Search Steps, Search String, Repositories, and Selection Procedures

Step 1–Automatic search: This step relied on search strings and search engines. The main terms in the string were "*mutation testing*" and "*cost reduction*," which were used to compose the full search string as follows:

> ("mutation testing" or "mutation analysis" or "mutant analysis") **and** ("cost reduction" or "sufficient operator" or "sufficient mutation" or "constrained mutation" or "selective mutation" or "weak mutation" or "random selection" or "random mutation" or "random mutants" or "equivalent mutant" or "equivalent mutation")

The search string, sometimes customized for specific databases, was submitted to the search engines of the following databases:[3] IEEE Xplore,[4] ACM Digital Library,[5] Elsevier ScienceDirect,[6] Springer SpringerLink,[7] and Wiley Online Library.[8] Additional databases are also sources of primary studies in the snowballing and author survey steps, as described next.

Step 2–Snowballing: The backward snowballing technique [188] was applied to the set of studies selected in Step 1. For each selected study, we analyzed the list of references to identify other studies of interest, according to the inclusion and exclusion criteria.

Step 3–Author survey: We contacted every author of a selected study from the previous steps and asked for suggestions of additional studies. We analyzed their suggestions according to the inclusion and exclusion criteria.

Step 4–Post-publication update: Our set of selected studies was updated after our preliminary publication [54]. Particularly, we analyzed studies published in the 13th International Workshop on Mutation Analysis (Mutation'18), 11th International Conference on Software Testing, Verification and Validation (ICST'18), and 40th International Conference on Software Engineering (ICSE'18). The updated dataset also includes studies suggested by authors previously contacted in our author survey in step 3.

General study selection procedures: For each search step, one researcher performed the preliminary selection by reading specific parts of the retrieved studies (initially, title and abstract, and, if necessary, introduction and conclusion) to check whether they should be selected for full reading. In the final selection step, two researchers fully read each study and made the final selection decision, according to the inclusion and exclusion criteria. All

[3]The original repositories of the selected studies can be found in the references at the end of this paper as well in the companion website [55]. It is important to note that IEEE, Elsevier, and Springer can be considered "pure" (or private) study repositories since they only store and retrieve studies from their own database. ACM and Wiley, on the other hand, have "hybrid" characteristics, since their search engines retrieve studies from a variety of databases, thus overlapping results from other searches.

[4]http://ieeexplore.ieee.org – last accessed on July, 2019.

[5]https://dl.acm.org/ – last accessed on July, 2019

[6]http://www.sciencedirect.com/ – last accessed on July, 2019

[7]http://link.springer.com/ – last accessed on July, 2019

[8]http://onlinelibrary.wiley.com/ – last accessed on July, 2019

conflicts in the study selection were resolved by the two researchers in order to reduce potential threats.

3.4. Data Extraction, Synthesis, and Analysis

Data of interest were extracted and stored in customized forms. General data such as authors, study title, and year of publication were used to compose references. For this SLR, more specific pieces of information were necessary for the synthesis and analysis steps. The first consists of characteristics of the cost reduction techniques and the metrics used for cost reduction measurement. The second consists of the values obtained with respect to cost reduction savings and the achieved mutation scores.

4. Study Selection Results

This section summarizes our results at each study selection step and the final set of selected studies. Table 1 presents overall numbers for each step. Figure 1 shows the number of selected cost reduction studies per year from 1989 through (part of) 2018. Tables A.7 through A.13 in Appendix A summarize the studies in tabular form.

Table 1 presents the numbers of studies retrieved and selected from each repository and search round. The term *search round* (or *round*) is used to refer to an automatic search round, the snowballing step, the author survey, and the last update performed after our preliminary publication of partial results [54]. Notice that Table 1 displays only total numbers for the third, fourth, and fifth rounds since the studies were not retrieved from a particular database via automatic search.

Table 1: Number of studies per search round. Columns labeled with "Sel." include non-duplicated results for final selection. Columns labeled with "Retr." include duplicated search results. Note that the numbers of retrieved studies did not evolve consistently for Rounds 1 and 2 due to changes made in the search engines, and due to specifics of the grammar used to combine search terms in both rounds.

Search Round	Database / Search Engine												Total	
	IEEE		ACM		Elsevier		Springer		Wiley		Experts		..	
	Retr.	Sel.	Retr.	Sel.	Retr.	Sel.	Retr.	Sel.	Retr.	Sel.	Retr.	Sel.	Retr.	Sel.
1 - Autom	467	27	612	19	606	12	464	0	771	13	19	13	2939	84
(Apr, 2016)	8.88%	17.65%	11.64%	12.42%	11.52%	7.84%	8.82%	0.00%	14.66%	8.50%	0.36%	8.50%	55.89%	54.90%
2 - Autom	303	8	642	10	706	1	78	0	376	1	0	0	2105	20
(Oct, 2017)	5.76%	5.23%	12.21%	6.54%	13.42%	0.65%	1.48%	0.00%	7.15%	0.65%	0.00%	0.00%	40.03%	13.07%
3 - Snowb	-	-	-	-	-	-	-	-	-	-	-	-	72	17
(Oct, 2017)	-	-	-	-	-	-	-	-	-	-	-	-	1.37%	11.11%
4 - Survey	-	-	-	-	-	-	-	-	-	-	-	-	135	25
(Oct-Dec, 2017)	-	-	-	-	-	-	-	-	-	-	-	-	2.57%	16.34%
5 - Update													8	7
(Aug, 2018)													0.15%	4.58%
TOTAL	770	35	1254	29	1312	13	542	0	1147	14	19	13	5259	153
	14.64%	22.88%	23.84%	18.95%	24.95%	8.50%	10.31%	0.00%	21.81%	9.15%	0.36%	8.50%	-	-

All rounds were performed with the support of the StArt tool [73], which supports major steps of the SLR process, with particular help for the preselection and final selection phases. The first round was performed in April 2016 and resulted in 84 selected studies. The second round was performed in October 2017 and resulted in 20 additional studies. Snowballing (third round) was performed over the set of studies selected in the first search round, and resulted in 17 additional studies. The author survey (fourth round) targeted authors of the three previous rounds. It started on 24 October 2017, and responses arrived until 18 December 2017 (54 days). In total, 42 authors replied and provided 135 suggestions of additional studies. 25 additional, non-duplicate, studies were selected from this group. As described in Section 3.3, our dataset was updated based on the analysis of studies published in three events (Mutation'18, ICST'18, and ICSE'18), and with studies suggested by authors previously contacted in our author survey. This update added 7 new studies (*Search Round 5* in Table 1).

It is important to note that Table 1 only includes total numbers of non-subsumed studies, that is, it does not account for 22 studies that passed our inclusion criteria, but were subsumed (updated or extended) by more recent studies. References to the 22 subsumed studies can be found in Appendix A (Tables A.14 and A.15). Furthermore, as explained in Section 3.3, due to their "hybrid" nature, ACM's and Wiley's search engines retrieved results that overlap results from other queried databases. Therefore, Table 2 provides more precise information regarding the number of studies selected per database than Table 1.

Table 2: Number of studies per database.

Database	# of Studies		Database	# of Studies	
IEEE	58	37.91%	Elsevier	17	11.11%
ACM	33	21.57%	Wiley	20	13.07%
Springer	15	9.80%	Others	10	6.54%
Total				153	100.00%

Figure 1 shows the distribution of selected studies per year. Note that only part of 2018 is included. It is clear the number of studies has been growing since 2009. Analyses of selected studies are presented in the next sections.

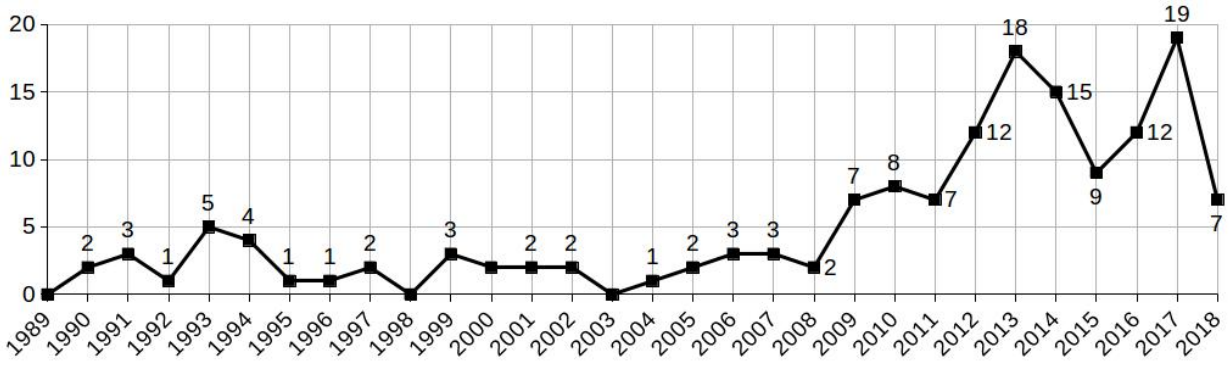

Figure 1: Distribution of studies per year. Only part of 2018 is included.

5. An Overall Characterization

This section presents and discusses an overall characterization of research into cost reduction of mutation testing. It starts in Section 5.1 by presenting the results of an initial classification based on the traditional "*do fewer*," "*do smarter*," and "*do faster*" categories [139].

In Section 5.2, we refine this classification into to a set of primary goals associated with cost reduction. Then, Section 5.3 describes techniques that have been developed to achieve those primary goals. Quantitative data are also presented in the following sections.

5.1. Traditional Classification: do fewer*,* do smarter*,* do faster

Offutt and Untch [139] grouped cost reduction techniques into three categories: *do fewer*, *do smarter*, and *do faster*. In their words:

> "*The 'do fewer' approaches seek ways of running fewer mutant programs without incurring intolerable information loss. The 'do smarter' approaches seek to distribute the computational expense over several machines or factor the expense over several executions by retaining state information between runs or seek to avoid complete execution. The 'do faster' approaches focus on ways of generating and running each mutant program as quickly as possible.*"

We classified the 153 selected studies according to Offutt and Untch's categories, as summarized in Figure 2. The bar labelled "1st Category" shows that 113 studies used a *do fewer* approach, 25 used a *do smarter* approach, and 15 used *do faster*. This bar reflects the primary goal of the 153 studies. However, some studies combined more than one goal. The bar labelled "2nd Category" shows that five studies used a second approach of *do fewer*, 50 had a second approach of *do smarter*, and six used *do faster*. For example, to automatically generate test cases, Papadakis and Malevris [148] used *control-flow analysis* supported by the *metamutants* technique. We classified this study as *do smarter* followed by *do faster*. To obtain the total number of studies that, for example, tried to *do fewer*, one can sum 113 (shown in the "1st Category" bar) with 5 (shown in the "2nd Category" bar).

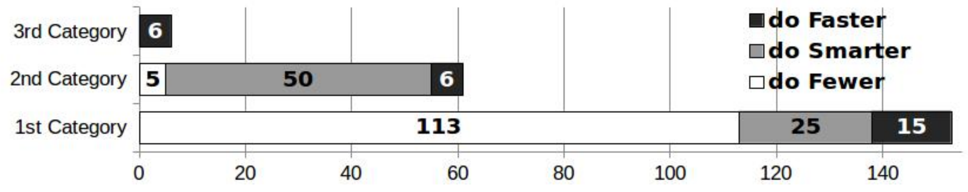

Figure 2: Number of studies per Offutt and Untch [139]'s categories.

Figure 3 shows the distribution of studies by year and category. We found more *do fewer* and *do smarter* approaches. For example, 17 out of 18 studies published in 2013 tried to run fewer mutants without significant effectiveness loss. In the same year, 11 out of 18 studies tried to reduce costs in a "smarter" way, usually by distributing computational expense, processing intermediate execution information from both the original program and its mutants, or by analyzing mutant execution statistics.

5.2. Primary Goals for Cost Reduction

The *do fewer*, *do smarter*, and *do faster* categories have been used for almost 20 years. However, as cost reduction techniques have become more complicated, this categorization

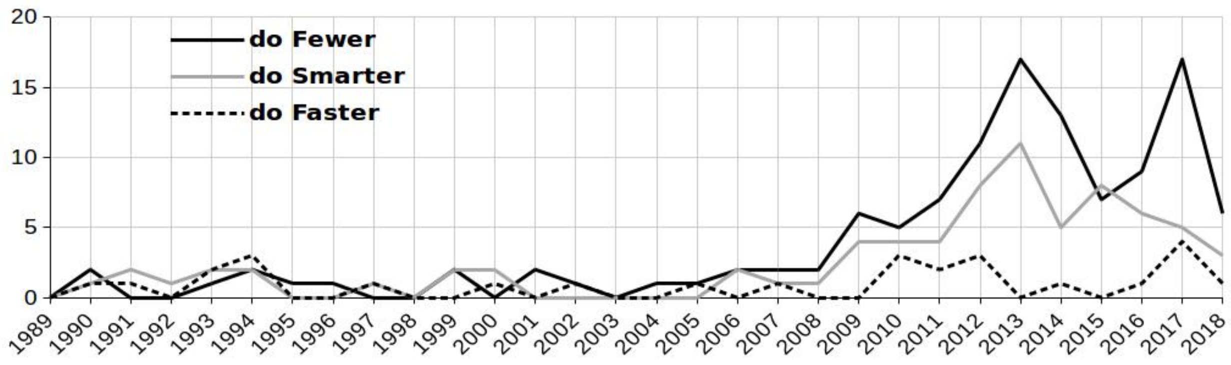

Figure 3: Distribution of studies per year for Offutt and Untch's [139] categories.

has become harder to apply. Some techniques fit into more than one category and others do not really fit into any of the three. Thus, a major contribution of this paper is a new, and modern, characterization of primary goals for cost reduction techniques.

Our analysis resulted in the six primary goals described as follows. Each is labeled with a symbol that is used as a marker in subsequent figures. We also provide a complete list of references to studies that pursued each primary goal.

PG-1: ($\bigtriangledown$) *Reducing the number of mutants:* The objective is to reduce the number of mutants that will be executed, preferably without reducing effectiveness [1, 3, 7, 9, 13, 19, 20, 21, 25, 28, 29, 30, 31, 33, 34, 39, 42, 43, 44, 45, 50, 64, 65, 71, 79, 81, 89, 91, 95, 101, 108, 109, 112, 117, 118, 126, 127, 128, 131, 138, 140, 141, 147, 153, 155, 156, 158, 160, 161, 165, 166, 171, 172, 174, 176, 177, 178, 180, 183, 186, 189, 190, 196, 197, 201, 202].

PG-2: ($\equiv$) *Automatically detecting equivalent mutants*: The objective is to automatically identify which mutants are equivalent to the original program, and then eliminate them from consideration [10, 31, 47, 56, 70, 74, 75, 97, 99, 100, 103, 118, 126, 129, 133, 134, 136, 142, 145, 154, 168, 170, 184].

PG-3: ($\triangleright\triangleright$) *Executing faster:* The objective is to reduce execution time by using novel algorithms, tool improvements, or special-purpose hardware. Some techniques analyze each mutant to decide whether it can be partially executed or even discarded without being executed [8, 22, 26, 27, 35, 40, 46, 47, 52, 57, 62, 63, 67, 80, 84, 86, 96, 107, 111, 115, 125, 133, 137, 150, 163, 164, 181, 182, 185, 187, 193, 195, 199].

PG-4: (ε) *Reducing the number of test cases or the number of executions:* The objective is either to find smaller test sets that are still as effective at killing mutants, or to identify groups of similar mutants to reduce the number of test runs [3, 12, 41, 43, 46, 59, 68, 71, 88, 92, 114, 140, 151, 167, 178, 198, 201, 202].

PG-5: ($\oslash$) *Avoiding the creation of certain mutants:* The objective is to define mutation operators or mutation generation algorithms to generate fewer mutants. The general idea is to generate only non-trivial (and non-equivalent) mutants [16, 32, 49, 53, 77, 78, 88, 90, 94, 98, 119, 175].

PG-6: (τ) *Automatically generating test cases:* The objective is to generate test cases automatically, to kill as many mutants as possible. Test case generation is typically guided by characteristics of the mutants, and can substantially reduce the effort required to create tests, which is usually done by hand [4, 6, 14, 15, 37, 38, 61, 70, 72, 113, 124, 130, 146, 148, 149, 150, 200].

Approaches that try to avoid the creation of certain mutants (*PG-5*) typically characterize circumstances during mutant generation that would lead to the creation of equivalent, trivial, and redundant mutants. These rules are then embedded into the code that implements the mutation operators to prevent such mutants from being created. Other approaches try to reduce the number of mutants (*PG-1*) post-creation. For these, the mutation operators are applied as originally specified; and the reduction of the mutant set is achieved either by constraining the set of operators or by strategically selecting mutants from the generated set (*e.g.* based on mutant characteristics or randomly). Clearly, achieving *PG-5* contributes to achieving *PG-1*. In other words, some of these goals can be seen as "side-effects" of others. Another example involves *PG-1*, *PG-4* and *PG-6*. Reducing the number of mutants (*PG-1*) implicitly reduces the number of test case executions (*PG-4*) and (possibly) the number of automatically generated test cases (*PG-6*). Figure 4 summarizes the relationships among the primary goals. Each edge indicates that the primary goal in the target node is indirectly achieved when the goal in the source node is achieved. For example, if PG-1 is achieved, then PG-6 is indirectly achieved.

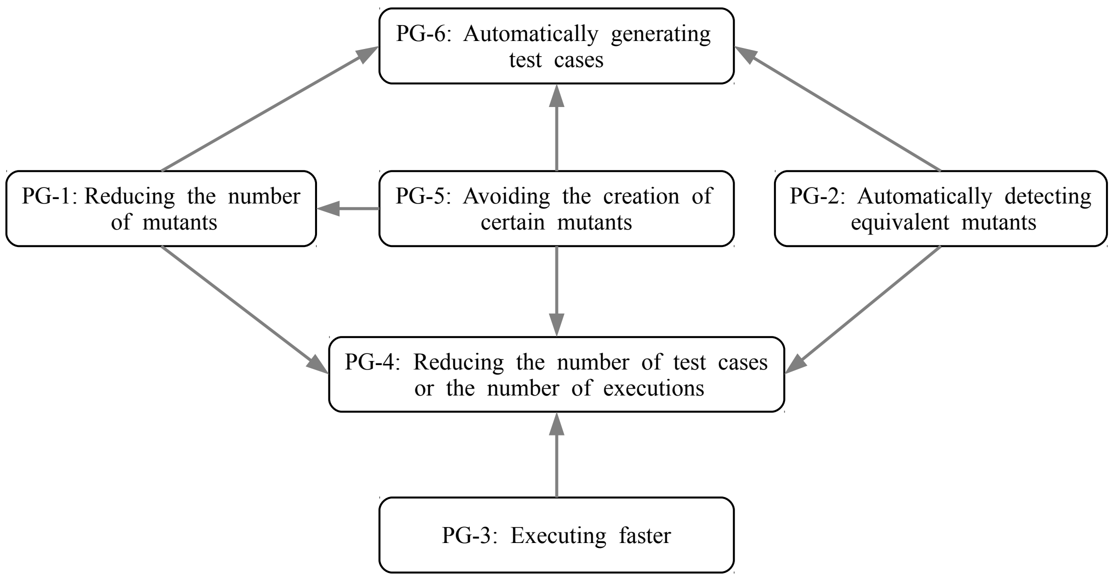
PG-6: Automatically generating test cases
PG-1: Reducing the number of mutants
PG-5: Avoiding the creation of certain mutants
PG-2: Automatically detecting equivalent mutants
PG-4: Reducing the number of test cases or the number of executions
PG-3: Executing faster

Figure 4: Relationships among primary goals.

We also emphasize that there is no clear disjoint, one-to-one, mapping between our new primary goals and Offutt and Untch's [139] categories. For instance, *PG-3* (*Executing faster*) can be achieved by (1) by using novel algorithms (*e.g.*, metamutants [182]), (2) improved tools (*e.g.*, compiler-integrated mutant generator and executor [35]), or (3) running mutants on special-purpose hardware (*e.g.* multi-processor machines [107, 137, 185]). These methods were classified by Offutt and Untch as *do smarter* (1 and 2) and *do faster* (3).

Figure 5 shows the number of studies that used each primary goal. The chart reveals that *reducing the number of mutants* was the most common, followed by *executing faster*, then *automatically detecting equivalent mutants*.

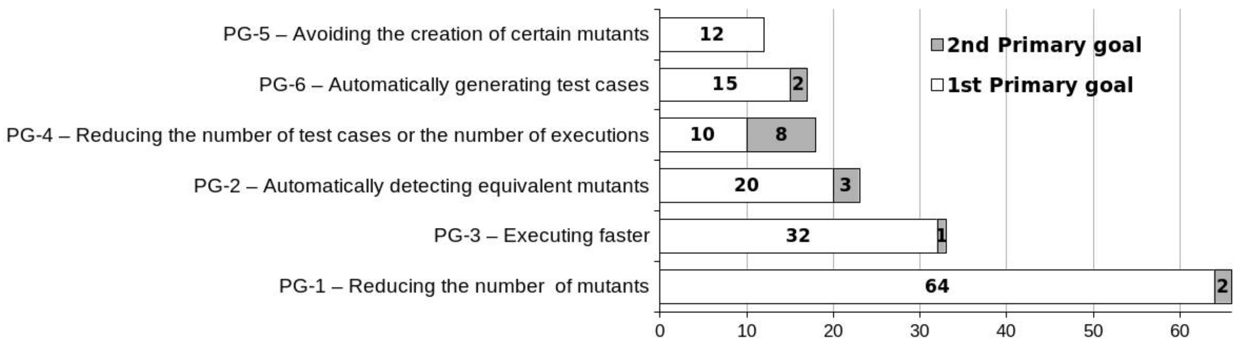

Figure 5: Number of studies per primary goal

5.3. Cost Reduction Techniques

This section presents a broad classification of cost reduction-related research according to 21 categories, or *techniques*. Some of the categories rely on mutation-specific research (such as *weak mutation* and *higher order mutation*), and others on classical software execution and analysis techniques (such as *parallel execution* and *control-flow analysis*). The timeline in Figure 6 shows the year each technique was introduced.

Five techniques appear twice in Figure 6: *random mutation* (1979 [2] and 1993 [122]); *higher order mutation* (1979 [2] and 2009 [158]); *weak mutation* (1982 [76] and 1990 [119]); *firm mutation* (1988 [192] and 2001 [80]); and *constrained mutation* (1991 [120] and 1993 [122]). These techniques were first described either without a focus on cost reduction [76, 192], in a high level way [120], or in a non-peer-reviewed publication [2], and were later published with more details in peer-reviewed studies [80, 119, 122, 158]. These initial references [2, 76, 120, 192] appear with different notation (dotted lines and smaller font size). Also, the techniques in Figure 6 are annotated with the symbols from the list of primary goals. For instance, *data-flow analysis* and *weak mutation* were first explored *to avoid the creation of certain mutants* (⊘) [119], whereas *metamutants* was first explored for *executing faster* (▷▷) [179].

The following list describes the 21 techniques. The descriptions are brief and omit details that were published in later years. Some categories have some overlap (including *selective mutation*, *sufficient operators*, *one-op mutation*, and *minimal mutation*), as discussed in Section 5.4.

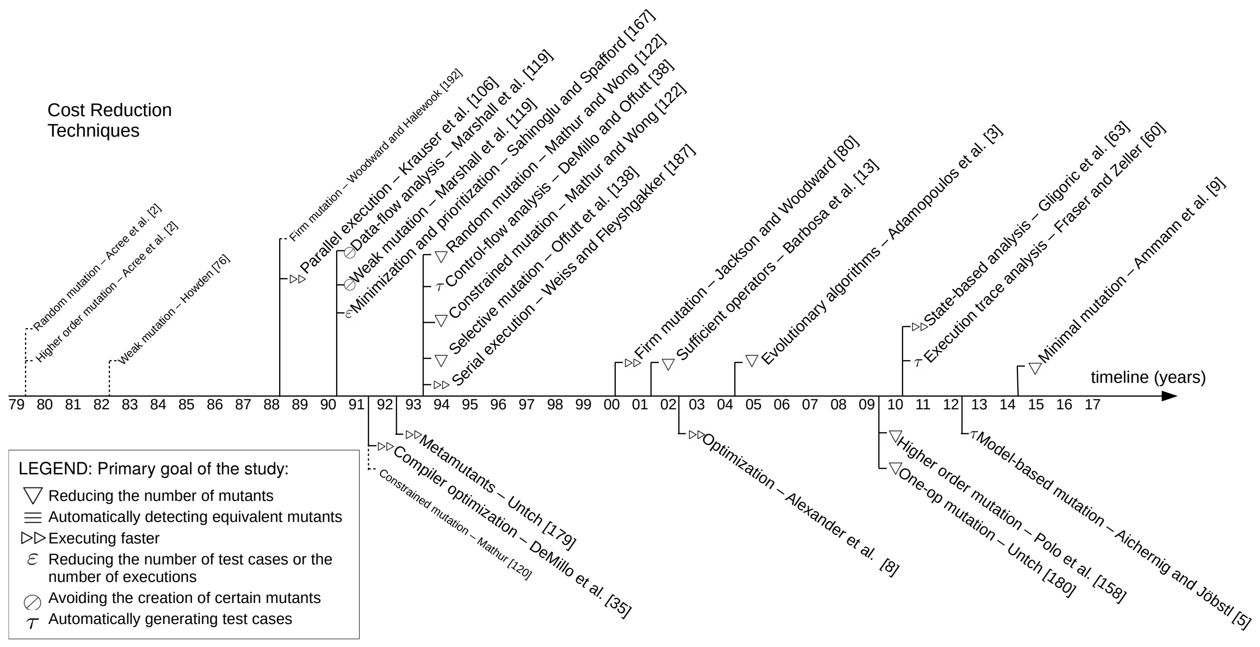
Cost Reduction Techniques
Random mutation – Acree et al. [2]
Higher order mutation – Acree et al. [2]
Weak mutation – Howden [76]
Firm mutation – Woodward and Halewook [192]
Parallel execution – Krauser et al. [106]
Data-flow analysis – Marshall et al. [119]
Weak mutation – Marshall et al. [119]
Minimization and prioritization – Sahinoglu and Spafford [167]
Random mutation – Mathur and Wong [122]
Control-flow analysis – DeMillo and Offutt [38]
Constrained mutation – Mathur and Wong [122]
Selective mutation – Offutt et al. [138]
Serial execution – Weiss and Fleyshgakker [187]
Firm mutation – Jackson and Woodward [80]
Sufficient operators – Barbosa et al. [13]
Evolutionary algorithms – Adamopoulos et al. [3]
State-based analysis – Gligoric et al. [63]
Execution trace analysis – Fraser and Zeller [60]
Minimal mutation – Ammann et al. [9]
timeline (years)
79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17
Metamutants – Untch [179]
Compiler optimization – DeMillo et al. [35]
Constrained mutation – Mathur [120]
Optimization – Alexander et al. [8]
Higher order mutation – Polo et al. [158]
One-op mutation – Untch [180]
Model-based mutation – Aichernig and Jöbstl [5]
LEGEND: Primary goal of the study:
▽ Reducing the number of mutants
≡ Automatically detecting equivalent mutants
▷▷ Executing faster
ε Reducing the number of test cases or the number of executions
⊘ Avoiding the creation of certain mutants
τ Automatically generating test cases

Figure 6: Timeline for introduction of cost reduction techniques in peer-reviewed studies.

T-1: *Random mutation*: This technique selects randomly from the complete set of mutants according to a predefined probability distribution. Random mutation has been interpreted as "choose X% of all mutants," "for each mutant, generate it with X% probability," and as "choose X% of mutants generated by each operator" [2, 19, 45, 65, 108, 122, 147, 153, 156, 190, 196, 197].

T-2: *Higher order mutation*: This technique combines two or more simple mutations to create a single complex mutant [1, 2, 42, 46, 68, 71, 92, 95, 101, 103, 112, 117, 141, 147, 158, 165, 166].

T-3: *Weak mutation*: This execution technique checks whether the state of the mutant is infected shortly after the mutated location has been executed, rather than checking the output after execution ends. If the state is infected, the mutant is killed immediately [47, 59, 76, 96, 101, 114, 119, 133, 150, 200, 201, 202].

T-4: *Firm mutation*: This is a variant of weak mutation where execution is allowed to proceed for some pre-defined duration after the state is infected [80, 133, 192].

T-5: *Parallel execution*: This technique executes mutants in parallel processors, reducing the total time needed to perform mutation analysis [27, 80, 107, 111, 137, 163, 185].

T-6: *Data-flow analysis*: This technique uses program data flow-related information to decide which mutants to generate and to analyze mutants. It considers whether variables that are more prone to failure during execution are reached and referenced [7, 32, 70, 74, 88, 98, 99, 103, 119, 129, 134, 142, 145, 155, 168, 170].

T-7: *Minimization and prioritization of test sets*: This technique analyzes the test suite to score test cases based on their effectiveness at killing mutants, then either eliminates test cases that are ineffective or runs the most effective test cases before the less effective test cases [41, 167, 178].

T-8: *Compiler optimization*: This technique uses compiler-related techniques to optimize mutant execution and analysis (for instance, to automatically detect equivalent mutants) [31, 35, 40, 86, 100, 129].

T-9: *Constrained mutation*: This technique chooses a subset of mutation operators to use. The choice relies on testers' intuition regarding the significance of particular groups of mutants [65, 120, 122, 156, 189, 190].

T-10: *Metamutants (or mutant schemata)*: This technique generates and executes mutants by embedding all mutants in one parameterized program called a *metamutant*. The metamutant is then compiled for fast execution. When run, the metamutant takes a parameter that tells it which mutant to run [67, 96, 114, 115, 148, 150, 164, 179, 181, 182, 187, 193].

T-11: *Control-flow analysis*: This technique uses program control flow-related information, focusing on execution characteristics to identify branches and commands that help determine which structures are most relevant to the generation and execution of mutants [14, 26, 32, 37, 38, 59, 70, 75, 81, 88, 89, 92, 103, 113, 118, 126, 130, 134, 136, 142, 145, 146, 148, 150, 151, 154, 155, 156, 168, 170, 177, 199, 200].

T-12: *Selective mutation*: This technique tries to avoid the application of mutation operators that are responsible for the most mutants or to select mutation operators that result in mutants that are killed by tests that also kill lots of mutants created by other operators. The idea is that if a test set, T_{op}, that is adequate for a subset of mutation operators op, also kills a very high percentage of all mutants, then we can select only the operators in op [20, 21, 31, 34, 44, 64, 65, 108, 127, 131, 138, 160, 174, 176, 178, 196, 197].

T-13: *Serial execution*: This technique dynamically determines classes of mutants that behave similarly, thus decreasing the number of mutants to be executed. The overhead incurred needs to be small relative to the time saved [57, 96, 114, 187].

T-14: *Sufficient operators*: This technique tries to determine an essential set of mutation operators by applying customized procedures [13, 29, 90, 109, 171, 172, 183, 197]. It can be seen as a special case of *selective mutation*, where the main difference is the complexity of the procedures applied to identify the final subset of mutation operators, which may be based, for example, on heuristics or statistics).

T-15: *Optimization of generation, execution, and analysis of mutants*: This technique groups approaches that reduce the cost of mutation testing by exploring strategies that did not fit other categories on this list. This category appears as *Optimization* in Figure 6 for brevity. More details are given later in this section [8, 10, 16, 22, 25, 26, 41, 43, 46, 52, 53, 56, 62, 67, 77, 78, 79, 84, 86, 90, 91, 94, 97, 125, 166, 175, 184, 186, 193, 195, 198, 199, 201, 202].

T-16: *Evolutionary algorithms*: This technique uses evolutionary algorithms to reduce the number of mutants, to reduce the number of test cases, or to identify equivalent mutants [1, 3, 12, 14, 15, 31, 33, 49, 50, 59, 61, 71, 72, 81, 124, 128, 140, 149, 150, 161, 174].

T-17: *One-op mutation*: This technique uses only a single mutation operator, which has the advantage of producing few mutants but also providing comprehensive coverage of the program [28, 30, 39, 42, 108, 180].

T-18: *Execution trace analysis*: This technique uses traces of execution of the original program and some mutants to decide which of the remaining mutants should be executed [61, 164].

T-19: *Model-based mutation*: This technique mutates formal or informal models of the program, and then uses the mutants to automatically generate test cases that are later used to kill mutants of the program [4, 6, 46, 47].

T-20: *State-based analysis*: This technique compares states of different mutant executions. When two mutants lead to the same mutation state, that is, when the same execution path is observed, only one needs to be executed and the result of the other can be inferred. Similarly, the technique creates groups of classes that define certain transition sequences such that only one needs to be verified [14, 62, 63, 84, 185].

T-21: *Minimal mutation:* This technique identifies and eliminates redundant mutants by applying the concepts of mutant subsumption and dominator mutants [9, 89, 108].

Note that *T-15* (*optimization of generation, execution, and analysis of mutants*) classifies studies that do not fit into any other category. For instance, Alexander et al. [8] implemented a mutation tool that uses *Java* reflection to mutate objects at run-time, thus avoiding the need to compile separate mutated programs. Durelli et al. [52] evolved the *Java* Virtual Machine to embed native support to speed up the execution of the original program and its mutants. Derezińska [41] proposed a clustering algorithm that uses results of mutant execution on subsets of test cases. A more recent example is Zhang et al.'s work [195], which used machine learning to devise predictive models to avoid the execution of some mutants. We classified these and other similar techniques as *optimization*-related. These are described in Section 5.5.

Figure 7 shows the distribution of studies per technique. Some studies used two or more techniques (discussed further in Section 5.4). Studies that used more than one technique are listed in Figure 7 with counts. For example, *minimal mutation* (*T-22*) has two units for *Used as 1st Technique* and one unit for *Used as 2nd Technique*. The amount of emphasis of each technique is based on the focus authors gave to that particular technique. As an example, Marshall et al. [119] reduced mutation costs by applying *data-flow analysis* to support *weak mutation*. Thus, *data-flow analysis* was the main technique while *weak mutation* was secondary. In another example, Wong and Mathur [190] compared *random mutation* with *constrained mutation*, without any clear preference for either. In this case, assigning higher participation for a particular technique was arbitrary. Therefore, even though Figure 7 shows varied levels of participation for most techniques, both techniques might have had similar importance.

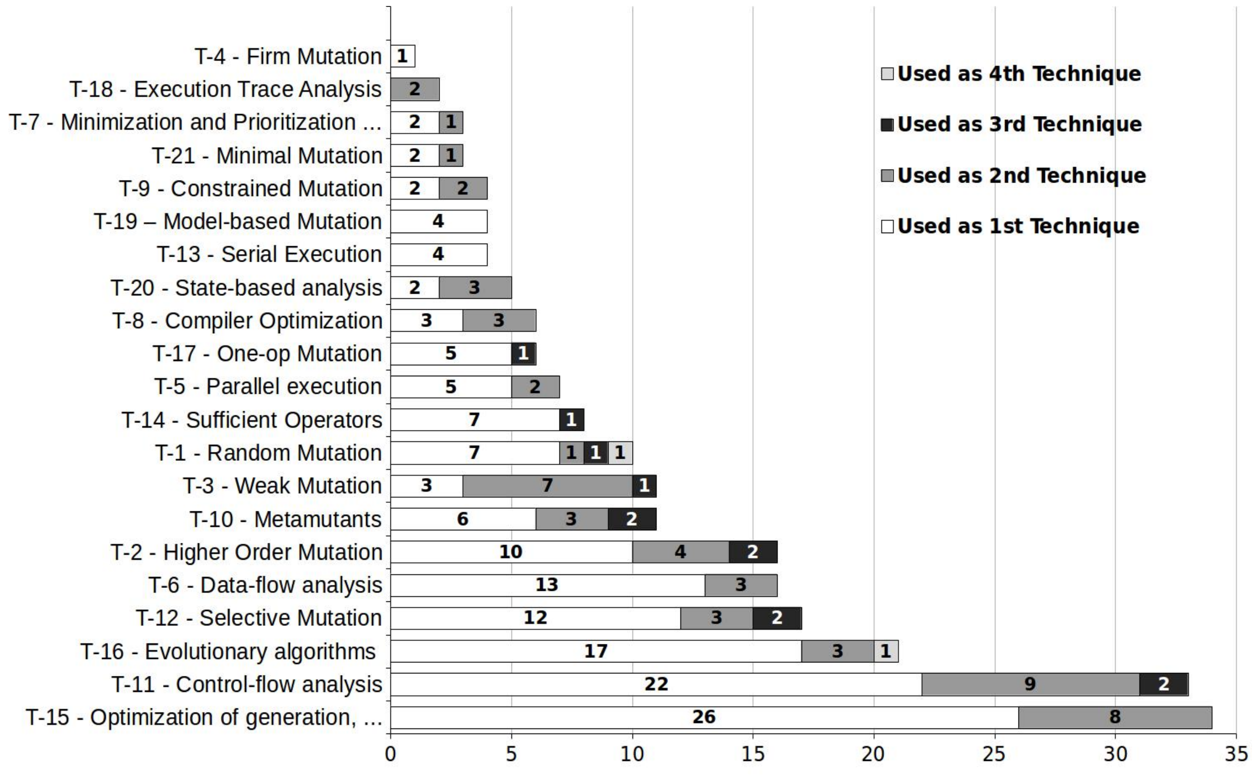

Figure 7: Number of studies per cost reduction technique.

Figure 8 shows the distribution of the nine most investigated techniques per year, split into three charts to improve readability. Even without statistical tests for time series, it is clear that researchers are studying some techniques more in recent years, such as *evolutionary algorithms*, *control-flow analysis*, *higher order mutation*, and *selective mutation*. Figure 9 illustrates this by showing the distribution of the six techniques most investigated over the five last years (2014-2018).

5.4. Additional Notes about the Cost Reduction Techniques and Primary Goals

As mentioned in Section 5.3, some of the techniques are similar. For instance, *constrained mutation* [120], *selective mutation* [138], *sufficient operators* [13], *one-op mutation* [180], and the recent *minimal mutation* technique [9] all reduce the number of mutants in some way, as far as possible without reducing effectiveness. After considering putting them all into

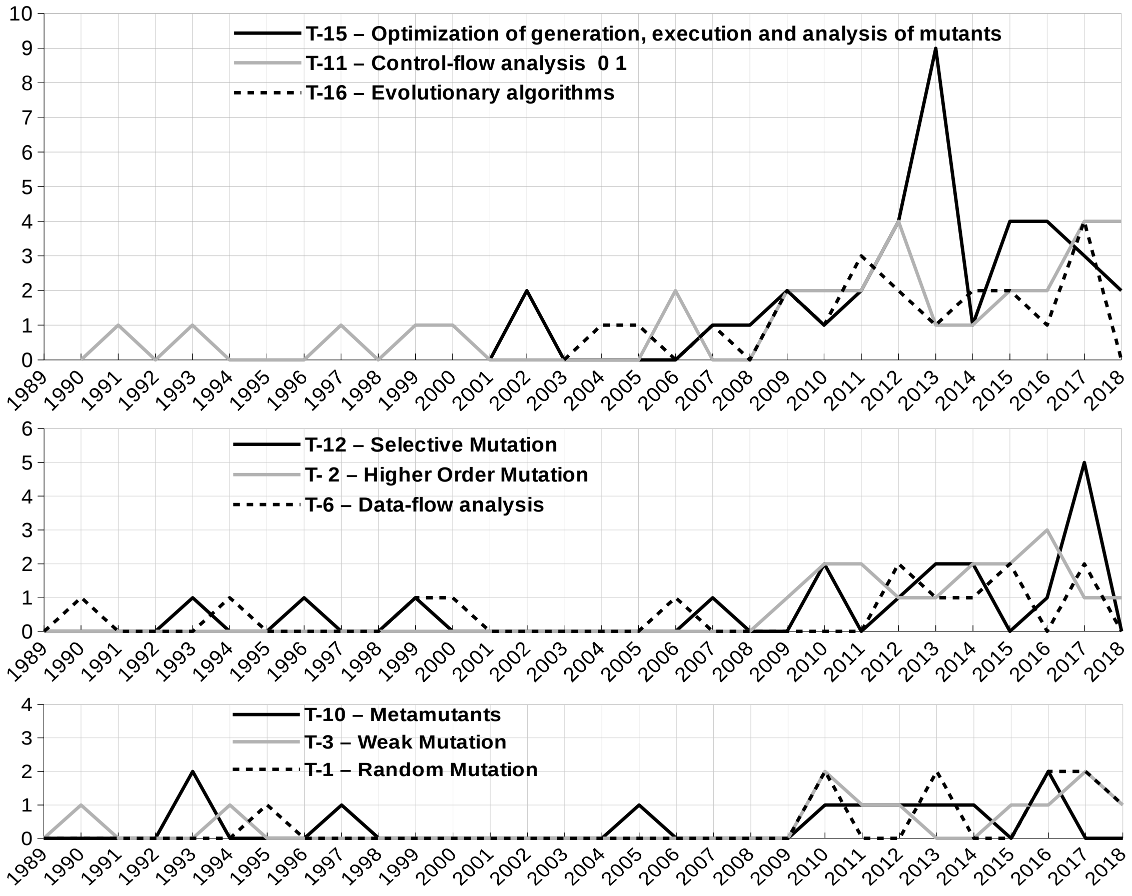

Figure 8: Distribution of studies of the top nine techniques per year.

one large category, we decided that a fine-grained categorization would be more helpful, and that small categories would be simpler to understand. Thus, we divided them on fairly specific characteristics. For example, *constrained mutation* relies on the testers' intuition, while *one-op mutation* relies on individual mutation operators. Along the same lines, we chose to separate *weak mutation* [76] and *firm mutation* [192], even though weak mutation could be viewed as a special case of firm mutation.

Another interesting observation is that the frequency of studies that combine techniques has increased over time. Figure 10 shows that techniques have been combined since research on cost reduction started, but this combined usage has increased since 2010. In 2009, two studies combined two techniques [81, 168], but in 2010 five studies combined two techniques [101, 147, 148, 174, 200], and one combined three [197]. By 2017, six studies combined two techniques [1, 32, 47, 89, 185, 201], and three combined three [14, 31, 65]. We also found a few studies that combined four techniques starting in 2011 [108, 150]. In total, of the 107 studies selected since 2010, 45 combined at least two techniques. Table 3 lists the

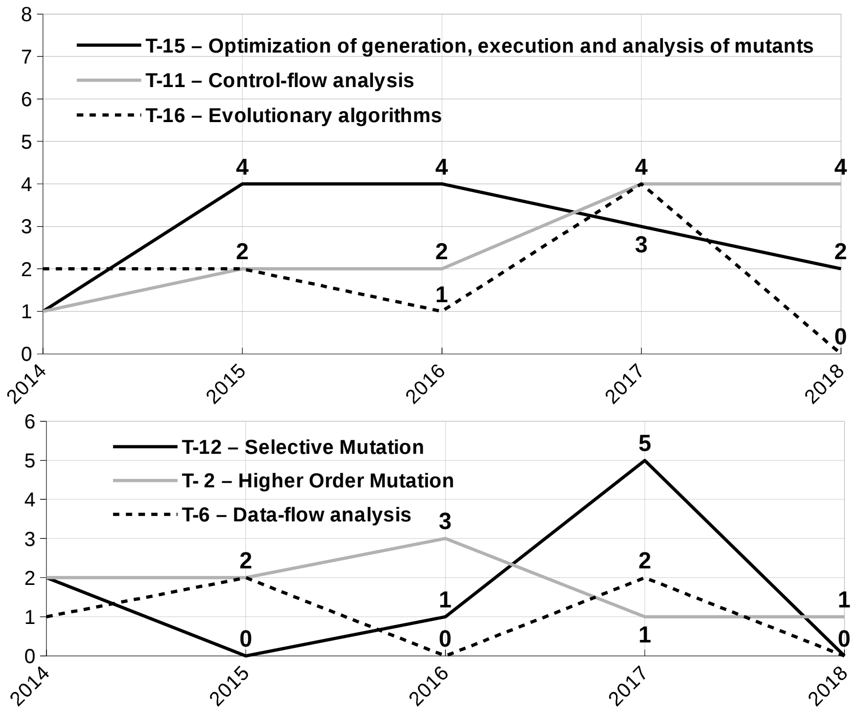

Figure 9: Distribution of studies of the top six techniques in the last five years.

studies that combined two or more techniques per year (the techniques are shown between parentheses).

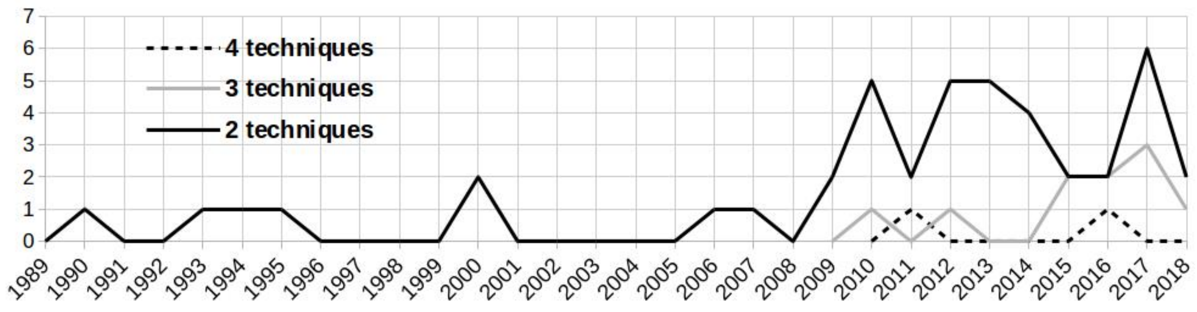

Figure 10: Distribution of studies that applied combined techniques per year.

Table 4 relates primary goals, cost reduction techniques, and studies. For each primary goal, the table lists all studies and the applied cost reduction techniques. The number of times a study is listed for a given goal corresponds to the number of applied techniques applied in that study, irrespective of which technique was applied to achieve a particular goal. For example, for *reducing the number of mutants* (*PG-1*), Gopinath et al. [65] applied

Table 3: List of studies that applied combined techniques per year (years with no studies are omitted).

Year	Studies and Respective Cost Reduction Techniques
1990	[119] (T-6 ,T-3)
1993	[187] (T-13 ,T-10)
1994	[129] (T-6 ,T-8)
1995	[190] (T-1 ,T-9)
2000	[70] (T-11 ,T-6); [80] (T-4 ,T-5)
2006	[134] (T-6 ,T-11)
2007	[178] (T-7 ,T-12)
2009	[168] (T-6 ,T-11); [81] (T-11 ,T-16)
2010	[197] (T-1 ,T-12 ,T-14); [101] (T-2 ,T-3); [147] (T-1 ,T-2); [174] (T-12 ,T-16); [148] (T-11 ,T-10); [200] (T-11 ,T-3)
2011	[86] (T-15 ,T-8); [92] (T-11 ,T-2); [150] (T-3 ,T-11 ,T-10 ,T-16)
2012	[96] (T-13 ,T-10 ,T-3); [62] (T-15 ,T-20); [199] (T-11 ,T-15); [88] (T-6 ,T-11); [155] (T-6 ,T-11); [61] (T-16 ,T-18)
2013	[145] (T-6 ,T-11); [196] (T-12 ,T-1); [170] (T-6 ,T-11); [193] (T-10 ,T-15); [41] (T-15 ,T-7)
2014	[84] (T-15 ,T-20); [142] (T-6 ,T-11); [71] (T-2 ,T-16); [164] (T-10 ,T-18)
2015	[166] (T-2 ,T-15); [103] (T-11 ,T-6 ,T-2); [59] (T-16 ,T-3 ,T-11); [90] (T-14 ,T-15)
2016	[114] (T-13 ,T-3 ,T-10); [108] (T-21 ,T-12 ,T-17 ,T-1); [67] (T-10 ,T-15); [46] (T-19 ,T-15 ,T-2); [42] (T-17 ,T-2)
2017	[185] (T-20 ,T-5); [89] (T-11 ,T-21); [201] (T-3 ,T-15); [14] (T-16 ,T-20 ,T-11); [47] (T-19 ,T-3); [65] (T-1 ,T-9 ,T-12); [32] (T-11 ,T-6); [31] (T-16 ,T-8 ,T-12); [1] (T-16 ,T-2)
2018	[26] (T-11 ,T-15); [202] (T-15 ,T-3); [156] (T-9 ,T-11 ,T-1)

three techniques: *t-1* (*random mutation*), *T-9* (*constrained mutation*) and *T-12* (*selective mutation*). Therefore, Gopinath et al.'s study is listed three times for *PG-1*.

Note that some studies tried to achieve two primary goals and applied two or more techniques. An example is by Just et al. [88]. That study applied techniques *T-6* (*Data-flow analysis*) and *T-11* (*Control-flow analysis*) to *reduce the number of test cases or the number of executions* (*PG-4*), and to *avoid the creation of certain mutants* (*PG-5*). In cases like this, the combination of techniques may allow one to achieve a combination of primary goals, therefore we do not distinguish between particular techniques applied to achieve particular goals.

Table 5 relates cost reduction techniques, primary goals, and studies. For each technique, the table lists all studies that addressed a particular goal. Similarly to Table 4, the number of times a study is listed corresponds to the number of primary goals it pursued times the number of techniques it applied (*e.g.* Just et al.'s [88] study is listed two times for *T-6* and two times for *T-11*).

The reader should notice that providing a complete list of tools that implement these cost reduction techniques is outside the scope of this paper. Readers could use information in this paper to identify the tools used in the experiments. Examples of useful information are the lists of references to studies with respect to primary goals (Section 5.2), techniques (Section 5.3), and the relationships between them (Tables 4 and 5).

5.5. Overview of Selected Studies

We describe each of the 153 selected studies in detail on the companion website to this paper [55][9]. We include a BibTeX file with an entry for every study, and a description field with more details than in this paper. The descriptions include the cost reduction goals of the studies (either explicit or implicit), and the achieved results. Every description also

[9]`http://goo.gl/edyF1n`–last accessed November 2018.

Table 4: Mapping between primary goals and cost reduction techniques.

Primary Goal	Cost Reduction Techniques and Respective Studies
PG-1 – Reducing the number of mutants	T-1 [45, 65, 108, 147, 153, 156, 190, 196, 197], T-2 [1, 42, 71, 95, 101, 112, 117, 141, 147, 158, 165, 166], T-3 [101, 201, 202], T-6 [7, 155], T-7 [178], T-8 [31], T-9 [65, 156, 189, 190], T-11 [81, 89, 118, 126, 155, 156, 177], T-12 [20, 21, 31, 34, 44, 64, 65, 108, 127, 131, 138, 160, 174, 176, 178, 196, 197], T-14 [13, 29, 109, 171, 172, 183, 197], T-15 [25, 43, 79, 91, 166, 186, 201, 202], T-16 [1, 3, 31, 33, 50, 71, 81, 128, 140, 161, 174], T-17 [28, 30, 39, 42, 108, 180], T-21 [9, 89, 108]
PG-2 – Automatically detecting equivalent mutants	T-2 [103], T-3 [47, 133], T-6 [70, 74, 99, 103, 129, 134, 142, 145, 168, 170], T-8 [31, 100, 129], T-11 [70, 75, 103, 118, 126, 134, 136, 142, 145, 154, 168, 170], T-12 [31], T-15 [10, 56, 97, 184], T-16 [31], T-19 [47]
PG-3 – Executing faster	T-2 [46], T-3 [47, 96, 133, 150], T-4 [80], T-5 [27, 80, 107, 111, 137, 163, 185], T-8 [35, 40, 86], T-10 [67, 96, 115, 150, 164, 181, 182, 187, 193], T-11 [26, 150, 199], T-13 [57, 96, 187], T-15 [8, 22, 26, 46, 52, 62, 67, 84, 86, 125, 193, 195, 199], T-16 [150], T-18 [164], T-19 [46, 47], T-20 [62, 63, 84, 185]
PG-4 – Reducing the number of test cases or the number of executions	T-2 [46, 68, 71, 92], T-3 [59, 114, 201, 202], T-6 [88], T-7 [167, 178], T-10 [114], T-11 [59, 88, 92, 151], T-12 [178], T-13 [114], T-15 [41, 43, 46, 198, 201, 202], T-16 [3, 12, 59, 71, 140], T-19 [46],
PG-5 – Avoiding the creation of certain mutants	T-3 [119], T-6 [32, 88, 98, 119], T-11 [32, 88], T-14 [90], T-15 [16, 53, 77, 78, 90, 94, 175], T-16 [49]
PG-6 – Automatically generating test cases	T-3 [150, 200], T-6 [70], T-10 [148, 150], T-11 [14, 37, 38, 70, 113, 130, 146, 148, 150, 200], T-16 [14, 15, 61, 72, 124, 149, 150], T-18 [61], T-19 [4, 6], T-20 [14]

Table 5: Mapping between cost reduction techniques and primary goals.

Primary Goal	Cost Reduction Techniques and Respective Studies
T-1 – Random Mutation	PG-1 [45, 65, 108, 147, 153, 156, 190, 196, 197]
T-2 – Higher Order Mutation	PG-1 [1, 42, 71, 95, 101, 112, 117, 141, 147, 158, 165, 166], PG-2 [103], PG-3 [46], PG-4 [46, 68, 71, 92]
T-3 – Weak Mutation	PG-1 [101, 201, 202], PG-2 [47, 133], PG-3 [47, 96, 133, 150], PG-4 [59, 114, 201, 202], PG-5 [119], PG-6 [150, 200]
T-4 – Firm Mutation	PG-3 [80]
T-5 – Parallel execution	PG-3 [27, 80, 107, 111, 137, 163, 185]
T-6 – Data-flow analysis	PG-1 [7, 155], PG-2 [70, 74, 99, 103, 129, 134, 142, 145, 168, 170], PG-4 [88], PG-5 [32, 88, 98, 119], PG-6 [70]
T-7 – Minimization and Prioritization of test sets	PG-1 [178], PG-4 [167, 178]
T-8 – Compiler Optimization	PG-1 [31], PG-2 [31, 100, 129], PG-3 [35, 40, 86]
T-9 – Constrained Mutation	PG-1 [65, 156, 189, 190]
T-10 – Metamutants	PG-3 [67, 96, 115, 150, 164, 181, 182, 187, 193], PG-4 [114], PG-6 [148, 150]
T-11 – Control-flow analysis	PG-1 [81, 89, 118, 126, 155, 156, 177], PG-2 [70, 75, 103, 118, 126, 134, 136, 142, 145, 154, 168, 170], PG-3 [26, 150, 199], PG-4 [59, 88, 92, 151], PG-5 [32, 88], PG-6 [14, 37, 38, 70, 113, 130, 146, 148, 150, 200]
T-12 – Selective Mutation	PG-1 [20, 21, 31, 34, 44, 64, 65, 108, 127, 131, 138, 160, 174, 176, 178, 196, 197], PG-2 [31], PG-4 [178]
T-13 – Serial Execution	PG-3 [57, 96, 187], PG-4 [114]
T-14 – Sufficient Operators	PG-1 [13, 29, 109, 171, 172, 183, 197], PG-5 [90]
T-15 – Optimization of generation, execution and analysis of mutants	PG-1 [25, 43, 79, 91, 166, 186, 201, 202], PG-2 [10, 56, 97, 184], PG-3 [8, 22, 26, 46, 52, 62, 67, 84, 86, 125, 193, 195, 199], PG-4 [41, 43, 46, 198, 201, 202], PG-5 [16, 53, 77, 78, 90, 94, 175],
T-16 – Evolutionary algorithms	PG-1 [1, 3, 31, 33, 50, 71, 81, 128, 140, 161, 174], PG-2 [31], PG-3 [150], PG-4 [3, 12, 59, 71, 140], PG-5 [49], PG-6 [14, 15, 61, 72, 124, 149, 150]
T-17 – One-op Mutation	PG-1 [28, 30, 39, 42, 108, 180]
T-18 – Execution Trace Analysis	PG-3 [164], PG-6 [61]
T-19 – Model-based mutation	PG-2 [47], PG-3 [46, 47], PG-4 [46], PG-6 [4, 6]
T-20 – State-based analysis	PG-3 [62, 63, 84, 185], PG-6 [14]
T-21 – Minimal Mutation	PG-1 [9, 89, 108]

includes the cost reduction technique as described in Section 5.3, and the technology and artifacts addressed (if provided by the original authors). As an example, the study of Offutt and Craft [129] is described in the complementary material as: "*Offutt and Craft [129] had the goal of detecting equivalent mutants. They used six techniques developed for compiler optimization. They analyzed the flow of data in mutants to identify equivalent mutants. In* *their experiment, they used 15 small Fortran programs and 14 mutation operators. The study automatically detected an average of 19.80% of the equivalent mutants.*"

The remainder of this section provides a more concise description of the selected studies, grouped by cost reduction goal and applied cost reduction technique. The groups of studies are organized chronologically, with a few exceptions when very interrelated studies are described within a given group (*e.g.* the studies by Domínguez-Jiménez et al. [50] and Delgado-Pérez et al. [33], which explored Evolutionary Mutation Testing to reduce the number of mutants to be executed). When possible, we established relationships between studies that extended prior studies. For the sake of completeness, studies that pursued two primary goals appear twice, each one in the subsection for each goal.

5.5.1. Studies that Tried to Reduce the Number of Mutants (PG-1)

Offutt et al. [138] extended and used the Mothra [37] tool to experiment with several *selective mutation* options on 10 *Fortran* programs. Offutt et al. [131] extended their prior work [138] by establishing a reduced set of operators (ABS, UOI, LCR, AOR, and ROR). Mresa and Bottaci [127] proposed a new type of *selective mutation* that takes into account the cost of detecting equivalent mutants. Tuya et al. [178] proposed mutation operators for *SQL*. They used *selective mutation* and *minimization and prioritization of test sets* to reduce the number of mutants and the number of test cases. Zhang et al. [197] empirically compared *random mutation* with *selective mutation* and *sufficient operators*. Results of experiments with 7 medium-sized *C* programs provided hints that 7%-random selection would achieve similar test effectiveness. Sridharan and Siami-Namin [174] used the same programs used by Zhang et al. to experiment with a Bayesian (*evolutionary*) approach to *select* the operators that generated mutants that were hard to kill. Derezińska and Rudnik [44] investigated *selective mutation* for *C#* programs. Gligoric et al. [64] applied *selective mutation* to concurrent *Java* programs. Zhang et al. [196] applied *selective mutation* and subsequently applied *random mutation*. Bluemke and Kulesza [20, 21] also explored *selective mutation* and a combination of tools for test case generation, mutant generation, and program analysis. Kurtz et al. [108] claimed that *selective mutation* must be specialized to avoid noise introduced by redundant mutants. Gopinath et al. [65] theoretically and empirically compared several cost reduction techniques (*selective mutation*, *constrained mutation*, and *random mutation*), and used statistical analysis to evaluate operator-based mutant selection. Praphamontripong and Offutt [160] applied a set of new mutation operators for Web applications and analyzed redundancy among operators. Delgado-Pérez et al. [31] explored the Trivial Compiler Equivalence technique [100, 143] in combination with *selective mutation* and *evolutionary algorithms*, and Delgado-Pérez et al. [34] applied *selective mutation* to 83 classes written in *C++*. Sun et al. [176] investigated mutation testing for *WS-BPEL* programs and used *selective mutation* to assess applicability and efficacy of the operators.

Wong et al. [189] used *constrained mutation* and explored varied strategies to compose subsets of operators, and Wong and Mathur [190] compared *constrained mutation* with *random mutation*. Recently, Petrović and Ivanković [156] described an approach for selecting *productive* mutants. These are, to some extent, similar to *constrained mutation* as originally proposed. It relies on the developers' understanding of the code for selecting either killable or equivalent mutants.

Papadakis and Malevris [147] empirically evaluated *random mutation* and *higher order mutation*. Bluemke and Kulesza [19] explored random mutation with eight *Java* classes and used various tools (MuClipse, CodePro, and some tailor-made tools). Parsai et al. [153] proposed to apply weights to randomly selected mutants and to focus on only acceptable mutants. Derezińska and Rudnik [45] explored random mutation together with equivalence partitioning for object-oriented (*C#*) programs.

Barbosa et al. [13] defined a procedure to identify *sufficient operators*. Vincenzi et al. [183] applied the same procedure [13] to empirically evaluate unit-level and integration-level mutation operators for *C* programs. Siami-Namin and Andrews [171] and Siami-Namin et al. [172] explored *sufficient operators* for *C* programs supported by statistical methods (linear regression analysis). Delamaro et al. [29] presented a new procedure for defining *sufficient operators*. The process iteratively includes mutation operators until 100% mutation score is achieved. Lacerda and Ferrari [109] applied Barbosa et al.'s [13] method in *AspectJ* programs.

Adamopoulos et al. [3] applied an *evolutionary algorithm* that relies on a fitness function to avoid equivalent mutants during the co-evolution process. Only mutants that are hard to kill and test cases that are good at detecting mutants are selected. Ji et al. [81] applied an *evolutionary algorithm* to support a domain reduction technique to identify mutant clusters, and then executed one mutant per cluster. Domínguez-Jiménez et al. [50] proposed Evolutionary Mutation Testing (EMT) to reduce the number of mutants. They used the GAmera tool to apply the approach to *WS-BPEL* projects. Delgado-Pérez et al. [33] also explored the EMT approach for *C++* programs. Nobre et al. [128] explored three multi-objective algorithms (namely, MTabu, NSGA-II, and PACO). Oliveira et al. [140] proposed a new genetic co-evolutionary algorithm and compared it with five other evolutionary approaches. Harman et al. [71] investigated strongly subsuming *higher order mutants* (SSHOMs) and *evolutionary algorithms* to reduce both the number of mutants and the number of test cases. Quyen et al. [161] explored mutation testing of *Simulink* models with *evolutionary algorithms*. Abuljadayel and Wedyan [1] applied genetic algorithms to kill *higher order mutants* for *Java* programs.

Polo et al. [158] explored *higher order mutation* for *Java* programs. Kintis et al. [101] experimented with *higher order mutation* and *weak mutation*, and compared results against strong mutation for *Java* programs. Kaminski et al. [95] suggested that generating only higher order logical mutants was sufficient. Omar and Ghosh [141] applied *higher order mutation* to *AspectJ* programs. Reales et al. [165] explored *higher order mutation* for *Java* programs by using the Bacterio tool and different algorithms. Madeyski et al. [117] implemented four strategies for *higher order mutation* of *Java* programs with support of the JudyDiffOP tool. Reuling et al. [166] proposed an approach to perform *higher order mutation* testing on feature models of software product lines (SPLs), trying to avoid redundant and equivalent mutants. Lima et al. [112] experimented with four strategies for *higher order mutation* and compared results with first order mutation. Derezińska [42] analyzed first order and *higher order mutation* based on statement deletion operators for *C#* programs, and presented a high-level analysis focused on the benefits of the deletion operators.

Sun et al. [177] applied *control-flow analysis* and proposed four strategies that focus on the most diverse mutants. Just et al. [89] proposed an approach for handling mutants based on abstract syntax tree analysis and on the *minimal mutation* approach, originally proposed by Ammann et al. [9]. *Minimal mutation* was also explored by Kurtz et al. [108], together with *selective mutation*, *one-op mutation*, and *random mutation*.

Marcozzi et al. [118] applied *control-flow analysis* to reduce the number of equivalent mutants based on proofs of logical assertions. McMinn et al. [126] analyzed paths in *SQL* expressions to remove ineffective mutants, including non-compilable, impaired, equivalent, and redundant mutants.

Kaminski and Ammann [91] reduced the mutant set related to logical expressions by *optimizing* mutant generation based on minimal DNF predicates. Wedyan and Ghosh [186] proposed an preliminary analysis of source code to prevent the generation of equivalent mutants for *AspectJ* programs. Zhu et al. [201] explored *weak mutation* and Formal Concept Analysis (FCA) to group similar mutants and test cases. Inozemtseva et al. [79] prioritized mutating parts of the programs that are most fault-prone based on fault history information; they used mutants generated by the PIT tool. Cachia et al. [25] proposed incremental mutation testing, which tests newly modified code only. Derezińska and Hałas [43] tried to *optimize* the generation and execution of mutants by applying syntax tree analysis and exploring runtime resources of the *Python* interpreter. Zhu et al. [202] evolved their previous work [201] and explored six compression techniques based on *weak mutation* to optimize strong mutation.

Untch [180] proposed the *one-op mutation* technique to compose a reduced set of mutants. Deng et al. [39] applied *one-op mutation* to *Java* classes. Delamaro et al. [30] designed new deletion-related mutation operators for C programs, and implemented them in the Proteum tool. In another study, Delamaro et al. [28] explored other mutation operators for *one-op mutation*.

Patrick et al. [155] used *control-flow analysis* and *data-flow analysis* to identify mutants that had little impact on the program output, and hence are harder to kill. Al-Hajjaji et al. [7] performed static *data-flow analysis* to select reduced sets of mutants for configurable systems.

5.5.2. Studies that Tried to Automatically Detect Equivalent Mutants (PG-2)

Devroey et al. [47] detected equivalent mutants in finite automata, *i.e.* behavioral models, by applying *weak mutation* and language equivalence concepts.

Offutt and Craft [129] performed *data-flow analysis* in mutants of *Fortran* programs and used six *compiler optimization* techniques to identify equivalent mutants. Kintis et al. [100] and Papadakis et al. [143] also explored *compiler optimization* to detect equivalent mutants. In particular, they proposed the TCE (Trivial Compiler Equivalence) technique to remove equivalent and duplicated mutants. Hierons et al. [74] used program slicing to create simple mutants for *C* programs. Harman et al. [70] explored program dependence (*data-flow* and *control-flow analysis*) to detect equivalent mutants by checking weakly killed mutants, and to generate test data automatically. Offutt et al. [134] specified heuristics, based on *data-flow analysis*, to avoid equivalent mutants for class-level mutation operators. Schuler et al. [168] applied the concept of dynamic invariants (derived via *control-flow* and *data-flow analyses*) and showed that invariant-violating mutants are less likely to be equivalent. More recently, Schuler and Zeller [170] applied the same concepts to classify mutants as killable or equivalent, and to provide hints about the most relevant mutants to be analyzed. Papadakis and Le Traon [145] and Papadakis et al. [142] empirically investigated Schuler and Zeller's [170] approach as a way to reduce the effects of equivalent mutants in mutation testing. Kintis and Malevris [99] investigated problematic data-flow patterns, through static analysis, to automatically identify equivalent mutants. Kintis et al. [103] relied on runtime information to detect equivalent mutants. They explored first and *high order mutation* and applied *control-flow* and *data-flow analyses* to classify killable and equivalent mutants.

Offutt and Pan [136] devised a constraint-based technique to detect equivalent mutants. They applied *symbolic execution* and heuristics to recognize infeasible constraints among rules to generate test cases–if the constraints cannot be satisfied, the mutant cannot be killed and thus is equivalent. Patel and Hierons [154] presented Interlocutory Mutation Testing (IMT) as a predicate-based *control-flow analysis* technique to automatically classify equivalent and killable mutants in programs that are non-deterministic and susceptible to coincidental correctness. Holling et al. [75] used static analysis and symbolic execution to classify mutants as being killable, equivalent, and "don't know." More recently, Marcozzi et al. [118] implemented a technique to identify equivalent mutants by proving the validity of logical assertions.

Vincenzi et al. [184] *optimized* the prediction of equivalent mutants by applying Bayesian-learning. The outcomes depend on the size of randomly-generated test sets. Anbalagan and Xie [10] and Ferrari et al. [56] implemented an approach to automatically detect equivalent mutants of pointcut expressions in *AspectJ* programs. Kintis and Malevris [97] introduced the concept of *mirrored mutants*, which are mutants that exhibit analogous behavior, such that identifying one mirrored mutant as being equivalent could help recognize other equivalent mutants.

5.5.3. Studies that Tried to Execute Faster (PG-3)

Devroey et al. [46] explored *higher order mutation*, *model-based mutation*, and *optimization* strategies to reduce the number of mutants of formal transition systems.

Offutt and Lee [133] explored *weak mutation* to speed up mutant execution and to automatically detect equivalent mutants. Papadakis and Malevris [150] applied *control-flow analysis* and *evolutionary algorithms* to support test case generation, and performed weak mutation to check the coverage of generated *metamutants*. Kim et al. [96] proposed optimizing test executions by avoiding redundant executions that were identified using *weak mutation*. Devroey et al. [47] detected equivalent mutants of finite automata (behavioral) models. They applied *weak mutation* together with language equivalence concepts, to detect equivalent mutants and speed up mutation analysis.

Krauser et al. [107] unified mutants for *parallel execution* on Single Instruction Multiple Data (SIMD) machines. Offutt et al. [137] tried similar ideas with Multiple Instruction Multiple Data (MIMD) machines. Similarly to Offutt et al. [137], Choi and Mathur [27] developed PMothra, which enabled *parallel execution* of mutants using a hypercube computer. Jackson and Woodward [80] introduced the parallel *firm mutation* technique for *Java* programs, using *Java* threads to increase execution speed. Reales and Polo [163] reported results from five algorithms for *parallel execution of mutants* and showed that the mutant execution cost is reduced proportionally to the number of nodes being used. Li et al. [111] explored mutation testing for *Ruby* programs with *parallel execution* in a cloud infrastructure.

DeMillo et al. [35] investigated a *compiler optimization* technique that created mutants by applying small patches at compile time. Just et al. [86] explored conditional mutation to reduce the time to generate and execute the mutants. Denisov and Pankevich [40] introduced a new tool for mutation testing based on the Low-Level Virtual Machine (LLVM), which compiles only the code fragments that are mutated.

Untch et al. [179, 182] introduced *metamutants* to compile and execute mutants faster. Untch et al. [181] later implemented a metamutant-based tool (TUMs) using Mutant Schema Generation (MSG). Weiss and Fleyshgakker [187] proposed a new algorithm for *serial mutation* of *metamutants* for fast mutant execution. They later invented a new algorithm for *serial execution* and analysis of mutants called Lazy Mutation Analysis (LMA). Ma et al. [115] applied the *metamutant* concept in the MuJava tool to generate mutants for compiled *Java* code. Wright et al. [193] explored four different approaches that combine *metamutants* and *optimization* techniques to speed up mutation testing of database schemas. Reales and Polo [164] proposed the MUSIC technique, which implements *metamutants* and *execution trace analysis* to speed up mutation. Gopinath et al. [67] explored the concepts of *metamutants* and execution *optimization* techniques to speed up mutant execution by avoiding multiple mutant compilation and redundant, partial test case executions.

Zhang et al. [199] used *control-flow analysis* and *optimization* techniques to speed up mutation testing during regression testing. Chen and Zhang [26] also applied regression testing ideas, reporting fewer test cases required, fewer mutants executed, and fewer test executions.

Alexander et al. [8] used *optimization* techniques to mutate *Java* objects. The study describes a multi-threaded tool that sped up mutation. Bogacki and Walter [22] used aspect-oriented programming concepts to perform mutation while executing the original program to try to reduce mutant compilation and execution time. Durelli et al. [52] modified the *Java* Virtual Machine to embed native support to speed up execution. Gligoric et al. [63] presented a method to efficiently explore the states of multithreaded programs. The approach was implemented by Gligoric et al. [62]. Just et al. [84] implemented *optimizations* in the Major mutation tool, focusing on infected states and propagation to output. Wang et al. [185] extended Just et al. [84]'s approach by using meta-functions to fork new processes to reduce redundant executions. Zhang et al. [195] predicted whether mutants would be killed before execution by using a model that relies on features of mutants, tests, and coverage measures. McMinn et al. [125] virtualized mutation testing of databases, thus reducing communication with the database.

5.5.4. Studies that Tried to Reduce the Number of Test Cases or the Number of Executions (PG-4)

Kaminski and Ammann [92] introduced the minimal-MUMCUT logic-based testing criterion based on *higher order mutation*, *control-flow analysis*, and fault hierarchies. Harman et al. [71] investigated strongly subsuming *higher order mutants* (SSHOMs) and *evolutionary algorithms* to reduce both the number of mutants and the number of test cases. Devroey et al. [46] explored *higher order mutation*, *model-based mutation*, and *optimization* strategies to reduce the number of mutants in formal transition systems. Gopinath et al. [68] reduced the number of test case executions by creating "supermutants," that combine several first order mutants (i.e., *metamutants*).

Fraser and Arcuri [59] investigated *weak mutation*, *evolutionary algorithms* and *control-flow analysis* for automatic generation of reduced test sets. Ma and Kim [114] devised a mutant clustering approach to execute only one mutant from a group of mutants that are weakly killed by a given test case. The approach generates *metamutants* and *serial* mutants to speed up execution. Zhu et al. [201] explored *weak mutation* (more precisely, state infection analysis) and Formal Concept Analysis (FCA) to group similar mutants and test cases. Zhu et al. [202] evolved their previous work [201] and explored six compression techniques to improve the efficiency of strong mutation based on *weak mutation*.

Tuya et al. [178] proposed mutation operators for *SQL*. They used *selective mutation* and *minimization and prioritization of test sets* to reduce the number of mutants and required test cases. Sahinoğlu and Spafford [167] addressed *minimization and prioritization of test sets* by applying a sequential statistical procedure based on prespecified thresholds. Derezińska [41] defined a mutant clustering approach for *C#* programs to reduce the numbers of mutants and tests.

Papadakis and Malevris [151] used control-flow analysis to select test cases. Just et al. [88] also used

Zhang et al. [198] *optimized* mutant execution by predicting, without running, which mutants can be killed based on historical test coverage.

Adamopoulos et al. [3] applied an *evolutionary algorithm* that used a fitness function to avoid equivalent mutants during co-evolution. Only mutants that were hard to kill and test cases that were good at detecting mutants were selected. Ayari et al. [12] proposed an evolutionary approach based on ant colony optimization to generate tests automatically. Oliveira et al. [140] explored genetic co-evolutionary algorithms to generate reduced sets of test cases that achieve higher mutation scores.

5.5.5. Studies that Tried to Avoid Creation of Certain Mutants (PG-5)

Marshall et al.'s [119] approach predicts the impact of mutations of program variables from the strong mutation and *weak mutation* perspectives, thus avoiding the creation of many mutants. Just et al. [88] used *data-flow analysis* to avoid creating redundant relational and conditional operator mutants. They also prioritized test cases based on *control-flow analysis*. Kintis and Malevris [98] prevented the generation of equivalent mutants by defining data flow patterns that reveal code locations that should not be mutated. Delgado-Pérez et al. [32] defined class-level mutation operators for *C++* programs and a set of restrictions (based on *data-flow analysis* and *control-flow analysis*) to avoid the creation of unproductive mutants (equivalent, duplicate, invalid, and trivial).

Just and Schweiggert [90] defined rules to avoid the creation of unnecessary operator mutants (conditional, unary, and relational) mutants by using *sufficient operators*, *optimization*, and prior results [88].

Steimann and Thies [175] explored behavior-preserving constraints (based on rafactoring rules) to *optimize* the generation of killable mutants. Hu et al. [77] evaluated static and dynamic nature of class-level mutation operators. They proposed new rules to avoid the creation of equivalent mutants. Kaminski et al. [94] theoretically analyzed and updated fault hierarchies for relational mutation operators (ROR), which eliminate redundancy among mutants. Their analysis showed that only three out of seven possible ROR mutants need to be generated. Inspired by Kaminski et al.'s [94], Iida and Takada [78] introduced the notion of mutant killable preconditions to identify redundant mutants in control-flow statements. Belli and Beyazıt [16] proposed an event-based approach named *k-Reg-based* that avoids creating equivalent and redundant mutants for models written in a regular grammar. Fernandes et al. [53] proposed a strategy to create rules to avoid useless (equivalent and duplicated) mutants. The rules discard those mutants right before they are generated.

Domínguez-Jiménez et al. [49] explored the use of genetic algorithms to avoid creating mutants for WS-BPEL compositions.

5.5.6. Studies that Tried to Automatically Generate Test Cases (PG-6)

Zhang et al. [200] used *control-flow analysis* to introduce mutant-killing constraints into the program under test, and explored *weak mutation* concepts to guide the generation of test inputs. Papadakis and Malevris [150] used *control-flow analysis* and *evolutionary algorithms* to generate test cases, and performed *weak mutation* to evaluate coverage on the generated *metamutants*.

Harman et al. [70] explored program dependence (*data-flow analysis* and *control-flow analysis*) to detect equivalent mutants by checking weakly killed mutants, as well as to automatically generate test cases.

Papadakis and Malevris [148] performed *control-flow* analysis of *metamutants* to generate better test cases.

DeMillo and Offutt [37, 38] used symbolic execution (*control-flow analysis*) to develop Constraint-Based Testing (CBT) to automatically generate test cases. CBT generated tests that satisfy the conditions needed to reach mutants and infect the program state after the mutant, and discarded tests that did not add to the mutation score. Later, Offutt et al. [130] evolved the CBT technique by replacing symbolic execution algorithms with dynamic symbolic execution in a technique they called dynamic domain reduction (DDR). Liu et al. [113] explored path-wise test data generation, claiming it was more efficient than symbolic evaluation. Papadakis and Malevris [146] applied path selection based on *control-flow analysis* to generate tests.

Bashir and Nadeem [14] proposed a combined fitness function for *evolutionary algorithms* that uses both *control-flow* and *state-based* properties of the mutants to generate test data automatically. Baudry et al. [15] explored the concept of ant colonies in *evolutionary al-*

gorithms to generate test cases. Papadakis and Malevris [149] used dynamic information from mutant executions to generate test data automatically. Fraser and Zeller [61] presented *uTest*, which generates unit tests for *Java* classes based on genetic algorithms and *execution trace analysis*. Henard et al. [72] explored search-based optimization methods to minimize the number of selected configurations in Software Product Line testing and to maximize the number of mutants killed. Matnei Filho and Vergilio [124] also tested Software Product Lines using a multi-objective evolutionary approach.

Aichernig et al. [6] combined *model-based* testing and mutation testing to automatically generate test cases. Their ideas can be used for several languages (*e.g. Prolog*, *Java* and *C*). They also developed a tool named *MoMut* [4] to generate test cases from UML models.

Regarding RQ1 (*Which techniques support cost reduction of mutation testing?*), we noticed that mutation-related costs can be reduced by applying a wide range of techniques, sometimes individually and sometimes in combination. Some of the most common techniques are traditional software analysis methods such as *control-flow analysis* and *data-flow analysis*. Others, such as *selective mutation* and *higher-order mutation*, are common only within the mutation testing field, whereas still others are widely used in CS and Math (*e.g. compiler optimization*, *evolutionary algorithms*, and *optimization-related* techniques). These observations lead us to conclude that mutation-related cost reduction research is more interdisciplinary than in the past. That implies that collaboration with researchers from other areas can be very productive and should be encouraged.

6. Cost Reduction Metrics and Results

Another contribution of this paper is the analysis of the benefits (in terms of cost savings) versus the loss of effectiveness (in terms of the mutation score or test quality). The analysis concerns research questions RQ2 and RQ3 (defined in Section 3.1) and starts with the characterization of metrics that have been used in the selected studies (Section 6.1). Then, Section 6.2 summarizes the main findings regarding results collected with those metrics.

6.1. Metrics and their Timeline

Table 6 lists metrics and studies that applied those metrics. The third column of the table, "*Intent*," shows the intended goal of the metrics. Notice that the intents of some metrics overlap. Metrics *M-6* (*Number of equivalent mutants automatically detected*), *M-11* (*Number of killable mutants automatically detected*), *M-12* (*Number of equivalent mutants generated*), and *M-17* (*Number of duplicated mutants automatically detected*) have similar intents, despite the fact that they collect different values from the sets of mutants. Also notice that the definition for each metric can be derived from its names and intent. For instance, *M-3* (*Mutant execution speedup*) can be defined as the time spent (in a given time unit) for mutant execution when compared to conventional execution (*i.e.* without applying a cost reduction technique).

Table 6: Metrics to measure the cost reduction of mutation testing.

ID	Metric name	Intent	Studies that collected the metric
M-1	*Number of test cases required to achieve mutant coverage*	Estimate the number of test cases that need to be created (automatically or manually) or executed.	[20, 25, 26, 28, 29, 30, 34, 41, 44, 45, 71, 81, 92, 112, 113, 140, 147, 151, 165, 167, 178, 189, 190, 191, 201]
M-2	*Number of mutants to be executed*	Estimate the number of mutants that need to be executed.	[7, 9, 13, 16, 19, 20, 21, 25, 26, 28, 29, 30, 32, 34, 39, 41, 42, 43, 44, 45, 50, 64, 65, 71, 72, 78, 79, 81, 88, 89, 90, 91, 91, 94, 95, 109, 112, 117, 118, 119, 128, 131, 138, 140, 141, 147, 153, 158, 160, 161, 165, 166, 171, 172, 174, 175, 176, 178, 180, 183, 185, 190, 191, 196, 197, 201, 202]
M-3	*Mutant execution speedup*	Estimate how much time is required for mutant execution.	[22, 26, 27, 32, 43, 45, 46, 52, 57, 62, 63, 67, 84, 88, 90, 96, 107, 111, 114, 115, 117, 125, 133, 137, 150, 163, 164, 181, 182, 185, 187, 193, 195, 196, 201, 202]
M-4	*Mutant execution efficiency*	Estimate the performance gains or losses when two or more execution configurations (e.g. processor and network infrastructure) are compared.	[137]
M-5	*Test generation speedup*	Estimate how much time is required for test generation.	[38, 113]
M-6	*Number of equivalent mutants automatically detected*	Estimate the number of mutants that need be handled during mutant analysis.	[10, 47, 53, 56, 97, 99, 100, 118, 126, 129, 134, 136, 142, 145, 154, 168, 194]
M-7	*Number of cycles (or generations) in evolutionary algorithms*	Estimate the effort required to run a specific evolutionary algorithm.	[14, 15]
M-8	*Mutant compilation speedup*	Estimate how much time is required for mutant compilation.	[32, 53, 86, 115]
M-9	*Mutant generation speedup*	Estimate how much time is required for mutant generation.	[50, 86, 96, 115]
M-10	*Completeness of generated test suite*	Estimate the effort required to evolve the current set suite to achieve the intended mutant coverage.	[91, 124, 146, 151, 200]
M-11	*Number of killable mutants automatically detected*	Estimate the number of mutants that need be handled during mutant analysis.	[47, 75, 154, 168]
M-12	*Number of equivalent mutants generated*	Estimate the number of mutants that need be handled during mutant analysis.	[28, 29, 30, 39, 77, 95, 98, 101, 117, 147, 158, 186]
M-13	*Number of cycles in test generation algorithms*	Estimate the effort required to run a specific test generation algorithm.	[148]
M-14	*Mutant analysis speedup*	Estimate how much time is required for mutant analysis.	[6, 47, 117, 175, 194]
M-15	*Probability to kill mutants*	To obtain a probability for mutant coverage based on the available test set.	[155]
M-16	*Number of test case executions*	Estimate the number of test case executions for a given mutant set.	[14, 26, 43, 68, 96, 114, 164, 167, 198, 199]
M-17	*Number of duplicated mutants automatically detected*	Estimate the number of mutants that need be handled during mutant analysis.	[53, 100, 126]
M-18	*Variation in mutation score*	Estimate gains and losses with respect to mutation score.	[177]

Figure 11 lists all identified metrics in a timeline. The first studies that used the metrics were published in 1990 [119, 167] and 1991 [107]. Marshall et al. [119] estimated how many mutants would be executed (*M-2*) after using static data-flow analysis to avoid creating certain mutants. Sahinoğlu and Spafford [167] applied statistical methods to estimate the number of test cases required to reach a threshold mutation score (*M-1*), as well as to estimate the number of test case executions (*M-16*). Both studies used small programs. Krauser et al. [107] ran simulation experiments to measure mutant execution speedup (*M-3*) on parallel machines. The timeline also shows that (i) few metrics (*M-1* to *M-6*, and *M-16*) were used in the early days (1990 to 1994), and (ii) many more metrics were used in the subsequent 14 years.

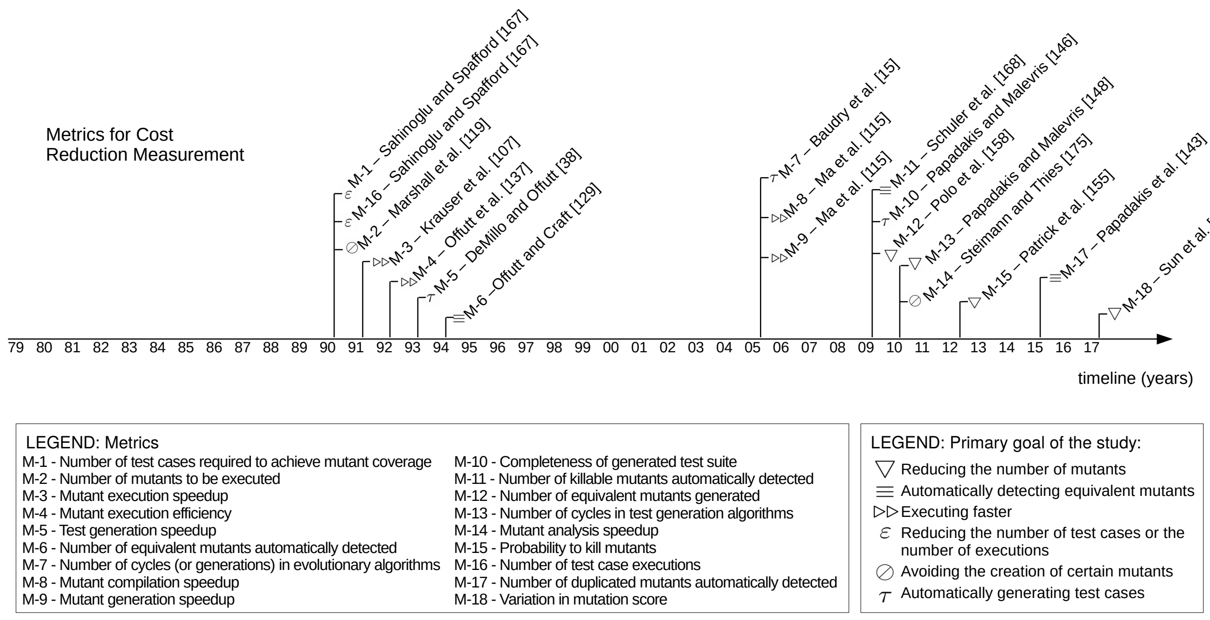
Metrics for Cost Reduction Measurement
ε M-1 – Sahinoglu and Spafford [167]
ε M-16 – Sahinoglu and Spafford [167]
⊘ M-2 – Marshall et al. [119]
▷▷ M-3 – Krauser et al. [107]
▷▷ M-4 – Offutt et al. [137]
τ M-5 – DeMillo and Offutt [38]
≡ M-6 – Offutt and Craft [129]
τ M-7 – Baudry et al. [15]
▷▷ M-8 – Ma et al. [115]
▷▷ M-9 – Ma et al. [115]
≡ M-11 – Schuler et al. [168]
τ M-10 – Papadakis and Malevris [146]
▽ M-12 – Polo et al. [158]
▽ M-13 – Papadakis and Malevris [148]
⊘ M-14 – Steimann and Thies [175]
▽ M-15 – Patrick et al. [155]
≡ M-17 – Papadakis et al. [143]
▽ M-18 – Sun et al. [177]
79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17
timeline (years)
LEGEND: Metrics
M-1 - Number of test cases required to achieve mutant coverage
M-2 - Number of mutants to be executed
M-3 - Mutant execution speedup
M-4 - Mutant execution efficiency
M-5 - Test generation speedup
M-6 - Number of equivalent mutants automatically detected
M-7 - Number of cycles (or generations) in evolutionary algorithms
M-8 - Mutant compilation speedup
M-9 - Mutant generation speedup
M-10 - Completeness of generated test suite
M-11 - Number of killable mutants automatically detected
M-12 - Number of equivalent mutants generated
M-13 - Number of cycles in test generation algorithms
M-14 - Mutant analysis speedup
M-15 - Probability to kill mutants
M-16 - Number of test case executions
M-17 - Number of duplicated mutants automatically detected
M-18 - Variation in mutation score
LEGEND: Primary goal of the study:
▽ Reducing the number of mutants
≡ Automatically detecting equivalent mutants
▷▷ Executing faster
ε Reducing the number of test cases or the number of executions
⊘ Avoiding the creation of certain mutants
τ Automatically generating test cases

Figure 11: Timeline for metrics used to measure cost reduction of mutation testing according to their uses in peer-reviewed studies.

Figure 12 shows the number of studies that used each metric. *Number of mutants to be executed* (*M-2*) was used the most (66 studies), followed by *Mutant execution speedup* (*M-3*) (36 studies) and *Number of test cases required to achieve mutant coverage* (*M-1*) (25 studies). If we consider only recent research, particularly studies published in the last five years (2014-2018), the most applied metrics are the same as in Figure 12. The distribution of studies per year is depicted in Figure 13.

Regarding RQ2 (*Which metrics are used to measure the cost reduction of mutation testing?*), we noticed that a variety of metrics have been used, with emphasis in measuring the *Number of mutants to be executed* (*M-2*), *Mutant execution speedup* (*M-3*), and the *Number of test cases required to achieve mutant coverage* (*M-1*). These are important factors related to the

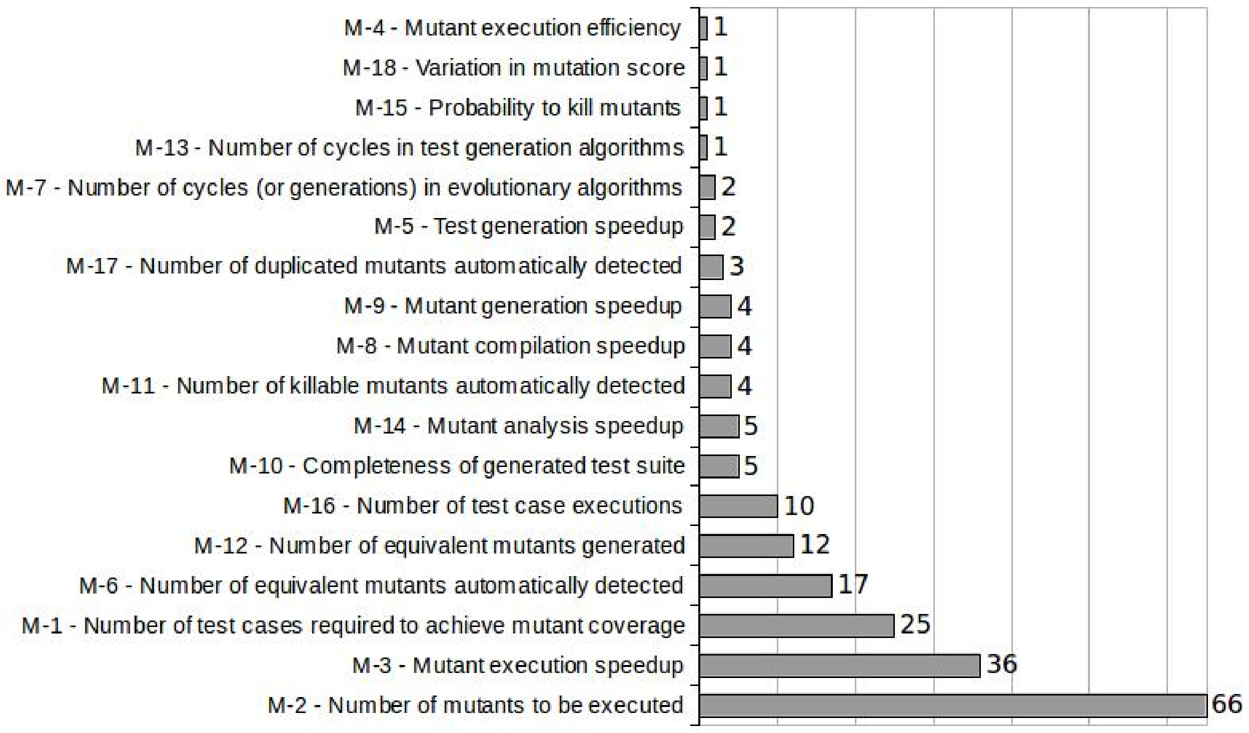

Figure 12: Number of studies per metric.

cost of mutation testing. Nonetheless, the results of our SLR for RQ2 also indicates that a major challenge for the practical adoption of mutation testing, *dealing with equivalent mutants*, has not been addressed as much as the others and we recommend more research into the problem. Specifically, only 11% (17 of 151) of the studies used metric *M-6* (*Number of equivalent mutants automatically detected*) and 8% (12 of 151) used *M-12* (*Number of equivalent mutants generated*).

Note about complex metrics not listed in this section: The list of metrics identified in this SLR is not intended to be complete, because several studies applied customized, and somewhat complex, metrics in the experiments. Examples can be found in the studies by Mresa and Bottaci [127], Delamaro et al. [28], and Delgado-Pérez et al. [34], and probably more. Mresa and Bottaci [127] estimated the cost of test case generation to be a combination of factors such as the number of mutant executions and the number of redundant tests. They also defined metrics such as *relative test generation cost* and *relative equivalence detection*. Delamaro et al. [28] devised a weighted cost function that is based on the *relative cost* of a mutation operator instead of being based on its absolute cost. Delgado-Pérez et al. [34] measured the *degree of redundancy* of a mutation operator, which is defined as the ratio of the number of redundant mutants generated by an operator to the number of mutants generated

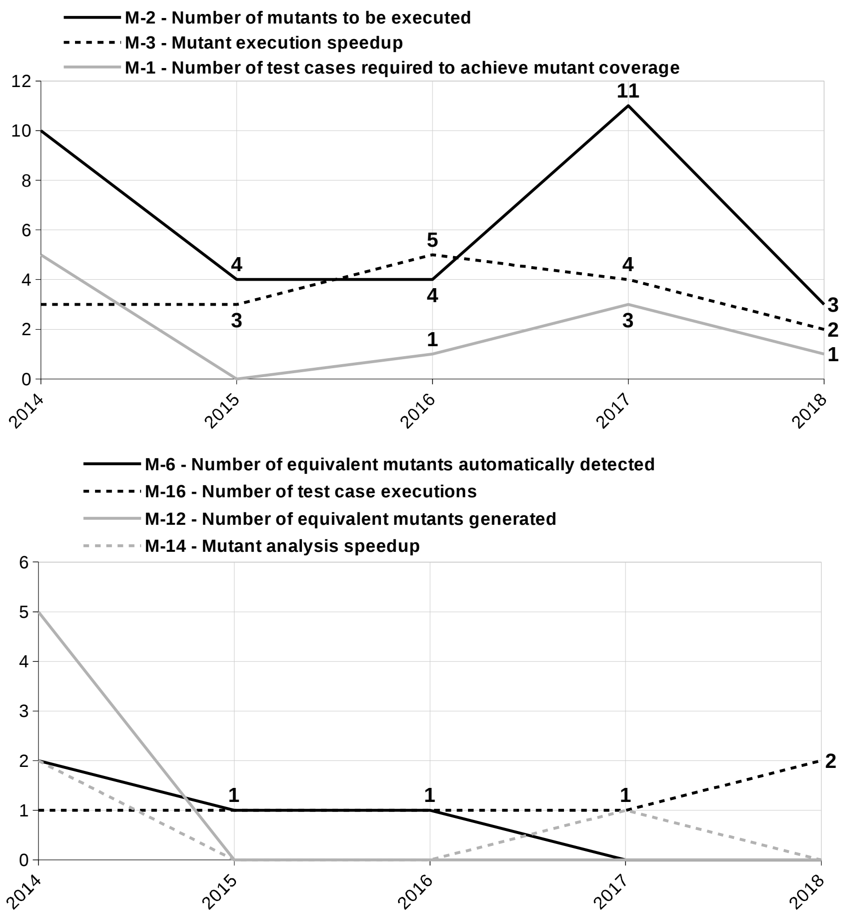

Figure 13: Distribution of studies that applied the top seven metrics in the last five years.

by the remaining operators. In our analysis, we focused only on identifying metrics that are derived from quantifiable elements directly extracted from the mutant creation, execution and analysis steps. Typical examples are the time saved to run a set of mutants (reflected by the *mutant execution speedup* (*M-3*) metric), and the size of mutation-adequate test sets (reflected by the *number of test cases required to achieve mutant coverage* (*M-1*) metric). The measurements used most often are presented in the next section.

6.2. Results About Cost Reduction Measurement

Figures 14–17 show the results achieved by different studies with respect to different cost reduction measurements. These data reflects the first part of RQ3: "*What are the savings (benefit) ... for the techniques?*". Results for the four most used metrics are displayed (*M-2*, *M-3*, *M-1*, and *M-6*), grouped by the three most common programming languages (*Fortran*, *Java*, and *C*). Figures 18 and 19 show the mutation scores achieved in the same studies listed in Figures 14 and 16 (*i.e.* for metrics *M-2* and *M-1*). These data address the second part of RQ3: "*What are ... loss of effectiveness (as proxied by mutation score) for the* *techniques?*". We do not present mutation score data for metrics *M-3* and *M-6*, since they were not reported for techniques applied to speed up mutant execution and automatically detect equivalent mutants. Furthermore, some values are not available so are missing in Figures 18 and 19.

In Figures 14 through 19, the bars show the minimum, mean, and maximum percentages of either cost reduction or mutation score. Note that not all values could be extracted from the studies. From some studies, we extracted only minimum and maximum values (for example, Wong and Mathur [190] in Figure 14.a). Sometimes we could extract only the mean values (for example, Vincenzi et al. [183] in Figure 14.b). Furthermore, no values were extracted at all for some studies. This is because of the level of details of reported results, characteristics of the applied cost reduction techniques, and customized experimental settings. This also meant we could not create whisker charts, which would require more detailed data than were presented in several studies. Also note that some studies appear twice in the charts of Figures 14–19. Authors of such studies applied more than one cost reduction technique, and performed individual measurements regarding cost reduction and mutation score. An example is by Wong and Mathur [190] (Figures 14.a and 18.a), who applied *random mutation* and *constrained mutation*. Additionally, Papadakis and Malevris [147] (Figures 14.b and 18.b) explored *random mutation* and *higher order mutation*, and Kintis et al. [101] (Figure 17.c) applied *higher order mutation* and *weak mutation*. Section 6.4 further discusses some characteristics of the measurements we retrieved.

Figure 14 shows the results of studies that measured cost reduction using metric *M-2* (*Number of mutants to be executed*) for *Fortran* (14.a), *C* (14.b), and *Java* (14.c) programs. Figure 15 shows the results regarding *M-3* (*Mutant execution speedup*), while Figures 16 and 17 show results for metrics *M-1* (*Number of test cases required to achieve mutant coverage*) and *M-6* (*Number of equivalent mutants automatically detected*). Notice that *M-1* was applied in a single study that targeted *Fortran* programs, therefore the chart is not shown in Figure 16.

6.3. Analysis and Discussion

Figures 14 through 17 reveal wide variations in cost reductions from both the inter-study or the intra-study perspectives. Variations among mutation scores (Figures 18 and 19) were less frequent, but still noticeable. Note that the inter-study perspective can provide an interesting baseline to compare results reported on groups of studies by different scientists who used the same cost reduction techniques. The intra-study perspective, on the other hand, provides a fine-grained notion of how varied results can be for different subjects of a particular study.

Regarding the *Number of mutants to be executed*, average reductions ranged from 26.70% [88] (Figure 14.c) to 99.77% [175] (Figure 14.c). With respect to *Mutant execution speedup*, the average reductions ranged from 25.00% [90] (Figure 15.c) to 97.78% [27] (Figure 15.a). For *Number of test cases required to achieve mutant coverage* and *Number of equivalent mutants automatically detected*, the averages ranged from 7.80% [29] (Figure 16.a) to 92.60% [172] (Figure 16.a), and from 1.78% [53] (Figure 17.c) to 90.51% [39](Figure 17.c).

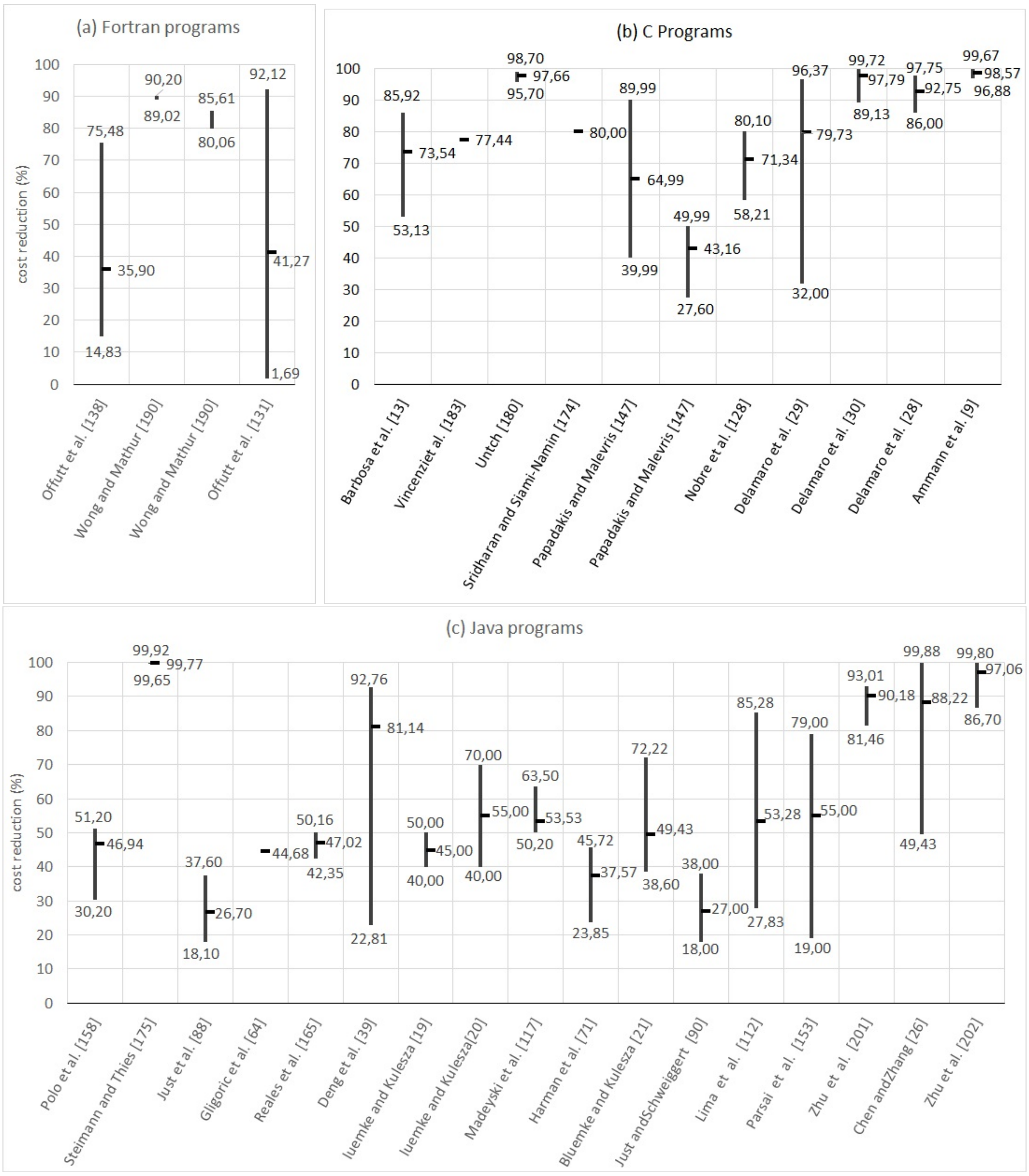

Figure 14: Cost reduction regarding *M-2 – Number of mutants to be executed*.

Regarding mutation scores, for both groups of studies (namely, the ones that measured *Number of mutants to be executed*, and *Number of test cases required to achieve mutant coverage*), the averages ranged from 0.800 [165] (Figure 18.c) to 1.000 [29] (Figure 18.b). Before

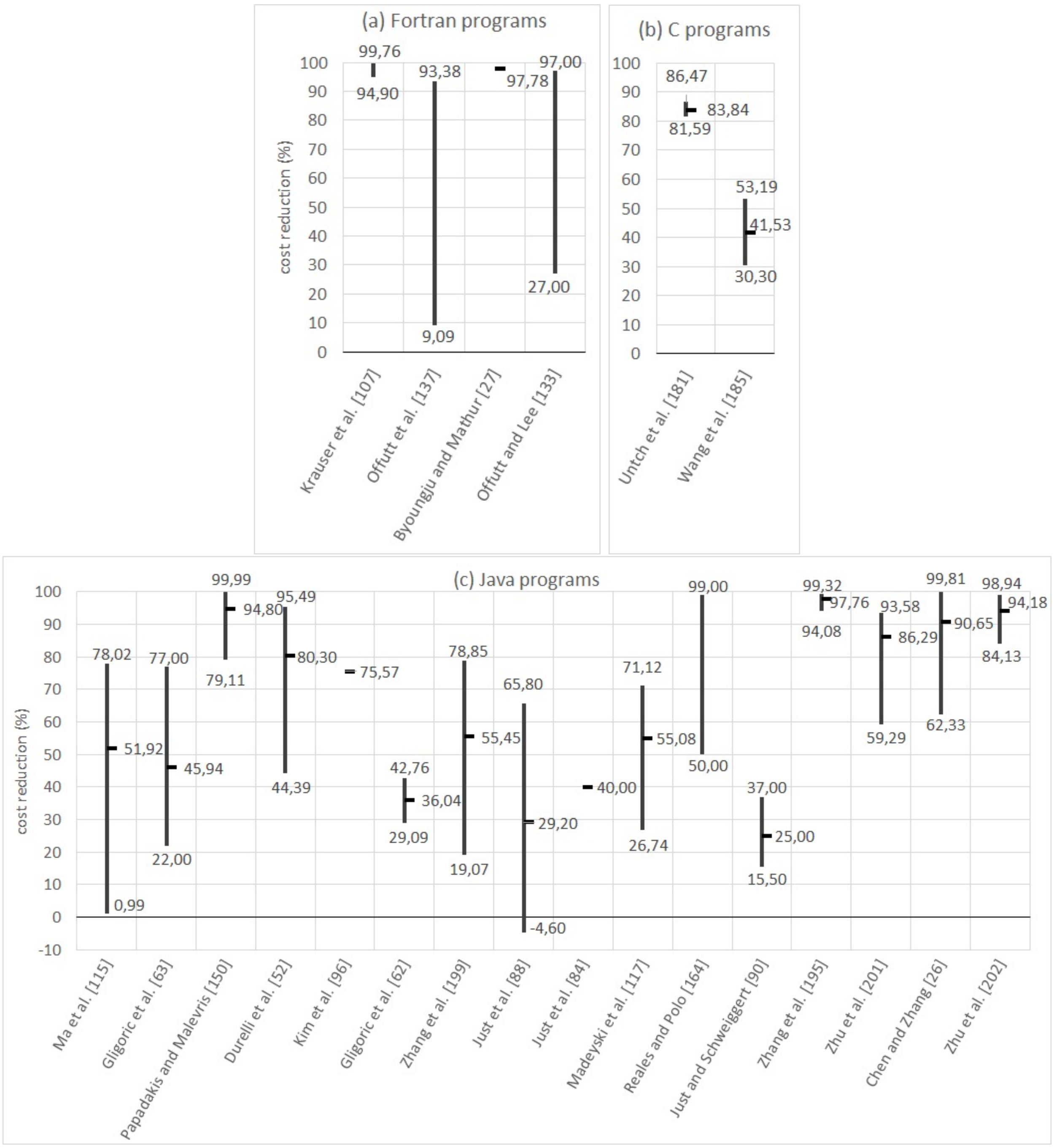

Figure 15: Cost reduction regarding *M-3 – Mutant execution speedup*.

choosing a cost reduction approach to adopt, it is important to consider the quality of the resulting test set regarding mutation-effectiveness, and hence its fault-revealing capability. Overall, the results presented in the charts show that while some studies achieved very high, even full, mutation score, some others did not satisfactorily maintain test effectiveness.

Taking an inter-study perspective reveals that the cost reduction variations presented in

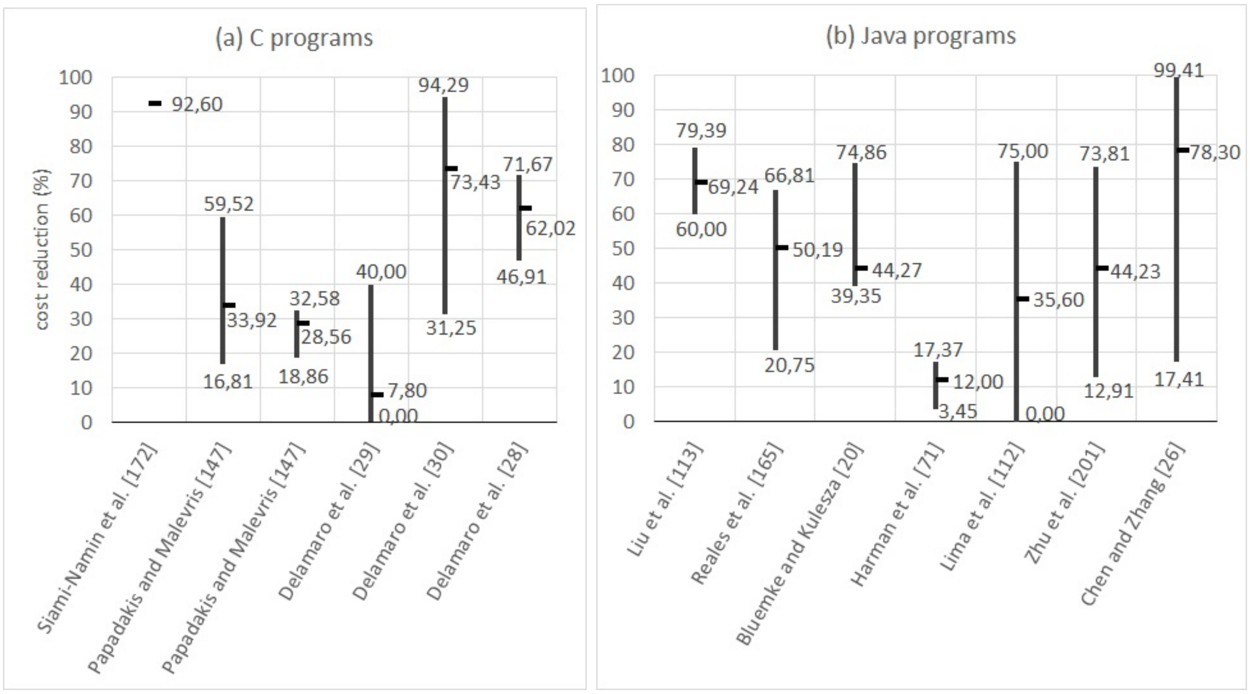

Figure 16: Cost reduction regarding *M-1 – Number of test cases required to achieve mutant coverage*.

Figures 14 through 17 are mainly due to the range of techniques that have been used, or to the experimental settings of studies that used the same technique and tools to a common set of programs. For example, the cost reductions shown in Figure 14.a were obtained by applying *selective mutation* [131, 138], *random mutation*, and *constrained mutation* [190]. These three studies targeted the same language (*Fortran*) with the same tool (*Mothra*). However, Offutt et al.'s studies [131, 138] addressed different *selective mutation* strategies (even though being applied to the same set of programs), while Wong and Mathur [190] used another set of programs and a different test generation tool.

Regarding RQ3 (*What are the savings (benefits) and loss of effectiveness (regarding mutation score) for the techniques?*), we found wide variations in results from both an inter-study and an intra-study perspective. Such variations, together with the number of metrics applied (see Section 6.1), make it harder to establish a baseline of studies for comparison in new experiments. Nevertheless, researchers can benefit from our results by using them as a reference to obtain more detailed information from specific studies, and to better design experiments that are comparable and reproducible.

6.4. Further Discussion about Results for Cost Reduction Measurement

This section presents more details about the measurements the studies used. These details affect how the results should be interpreted. We discuss six types of results: (i) results that represent close to (or even exactly) 100% cost reduction, (ii) results that depend on

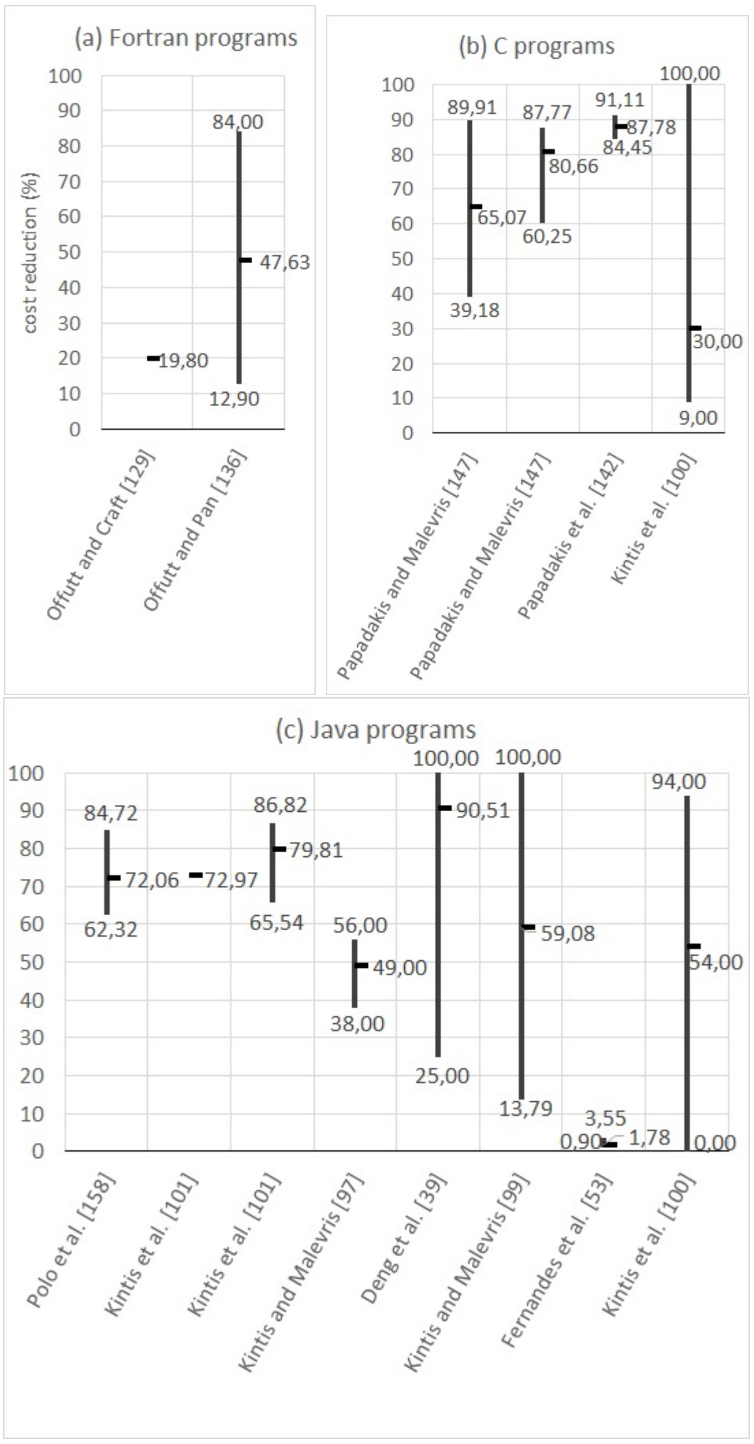

Figure 17: Cost reduction regarding *M-6 – Number of equivalent mutants automatically detected*.

either randomly or systematically defined thresholds, (iii) results that are very context-specific, (iv) imprecise results, and (v) results obtained from inconsistent subjects.

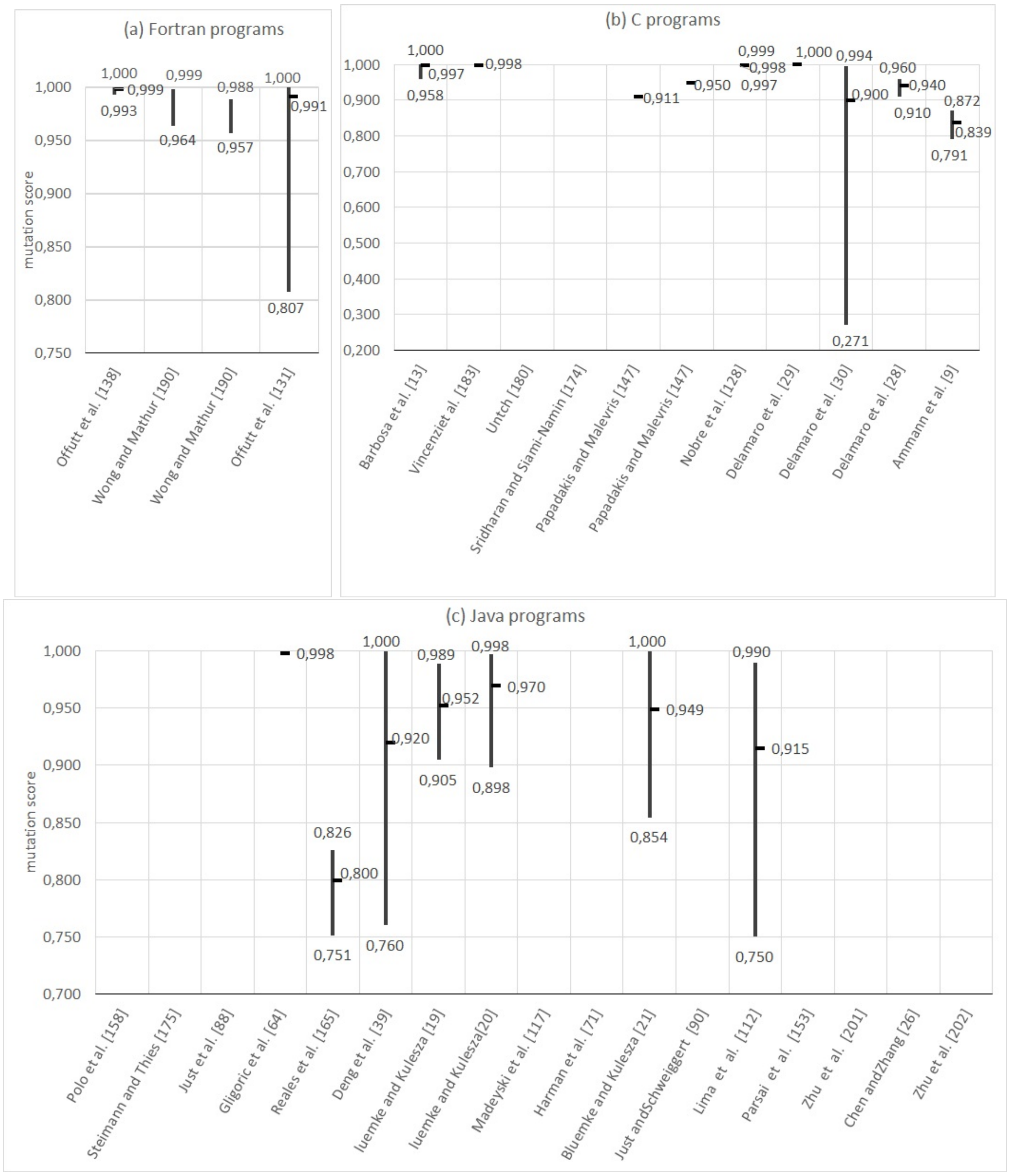

Figure 18: Mutation scores achieved in studies that applied metric M-2 – *Number of mutants to be executed*.

(i) *Results that are close to (or even exactly) 100% cost reduction*

Some studies reached 100% cost reduction or very close [10, 98, 146, 175]. Although appealing, some were not generalizable and some did not measure all cost factors. For instance, Anbalagan and Xie [10] automatically identified all equivalent mutants for a particular cat-

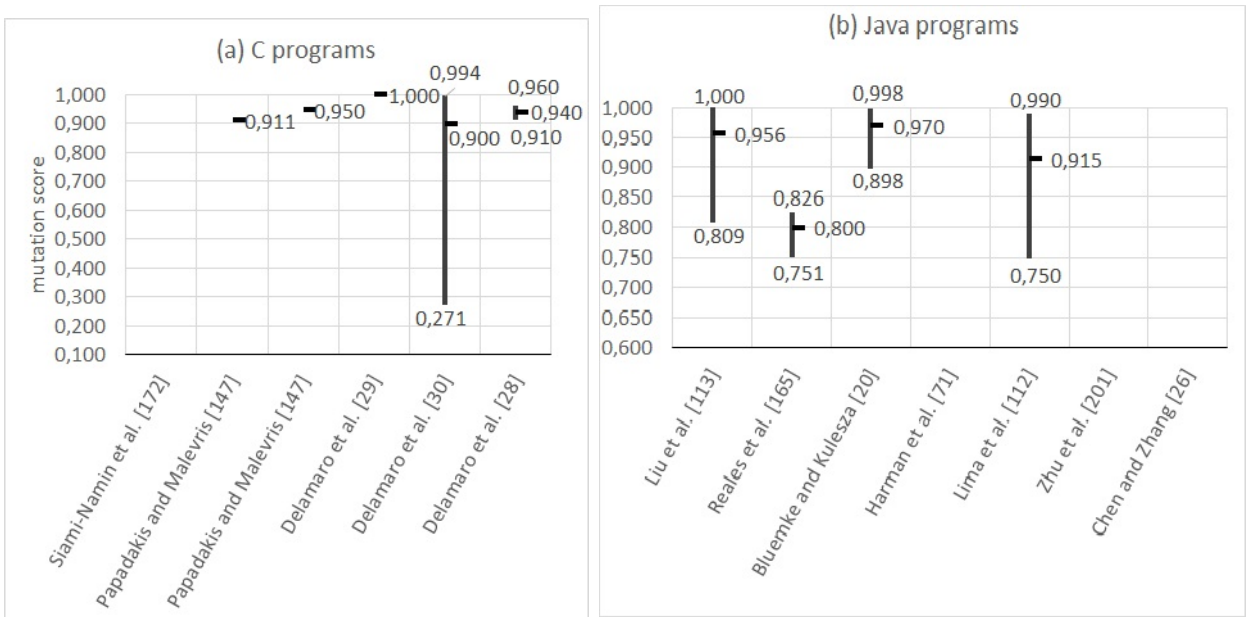

Figure 19: Mutation scores achieved in studies that applied metric M-1 – *Number of test cases required to achieve mutant coverage*.

egory of mutants: mutants of pointcut expressions of aspect-oriented programs written in the *AspectJ* language. Other costs that were not considered include the cost of running the code weaving compilation step (required by compilers of aspect-oriented languages), and the cost of running the mutant equivalence checking algorithm.

Another example is Papadakis and Malevris's [146] approach, which automatically generates tests to kill all non-equivalent mutants. That is, it produces mutation-adequate test suites. On the surface, this appears to eliminate the human cost of test data generation. However, the study did not include the manual mutant analysis cost, or the time to generate, compile, and execute mutants.

Steimann and Thies [175] reached 99.77% cost reduction using the *Number of mutants to be executed*, but only for a single type of mutant (access modifier change). Kintis and Malevris [98] reported cost reductions using *Number of equivalent mutants automatically detected* that widely ranged from 0 to 100% for *Java* programs. Patel and Hierons [154] investigated mutation testing for non-deterministic programs. The authors performed a preliminary evaluation based on a single program and identified 100% of equivalent mutants. Both studies omitted other costs.

(ii) *Results that depend on either randomly or systematically defined thresholds*

We did not present the cost reduction for some studies that used *evolutionary algorithms* (*T-16*), or for *optimization*-related techniques (*T-15*). This is because the results depended largely on the number of iterations of the algorithm, which is usually predefined arbitrarily based on factors such as timeout, achieved mutant coverage, and size of subjects. Some examples are the studies by Domínguez-Jiménez et al. [49], Fraser and Arcuri [59], and Reuling et al. [166].

Domínguez-Jiménez et al. [49] based their experiment on parameters such as population size and new individuals generated between different generations. Fraser and Arcuri [59] focused on generating tests to achieve mutant coverage by applying a combination of techniques (*evolutionary algorithms*, *weak mutation*, and *control flow analysis*). Their studies compared mutation-based test case generation with branch coverage-based tests. One evaluation compared their approach with conventional mutation testing. They found an increase in mutation score, but the execution of the test generation algorithm was limited to four minutes.

Reuling et al. addressed mutation testing for software product lines (SPL). The authors set up three levels of SPL feature combinations, and were not able to use subjects that had more than 200 features (they ran out of time). Zhang et al. [200] investigated automatic test generation for Java programs and set a timeout after a predefined execution time or number of iterations, so it was impossible to extract cost reduction and mutation score values.

(iii) *Results that are very context-specific*

Some studies not listed in Section 6.2 were done in very particular contexts. Examples include specific tools [6, 199, 200], regression testing [25, 26, 199] specific algorithms [163], and unusual sets of mutation operators [68, 92], or programming constructs [56, 186]. For instance, Zhang et al. [200] compared results from two test generation tools (*Pex* and *PexMutator*), but only *PexMutator* was designed for mutation testing, so there was no basis for comparison. Aichernig et al. [6] explored test case generation based on state machine models and measured cost reduction based on results produced by a prior implementation by the same authors [5][10]. Zhang et al. [199] studied mutation testing for regression testing of *Java* programs and compared their results with results produced by the *JAVALANCHE* tool [168].

Another example is by Reales and Polo [163], who studied *Mutant execution speedup*. The authors discussed results for five alternative parallel execution implementations, but they were not compared with sequential mutant execution.

Kaminski and Ammann [92] measured the *Number of test cases required to achieve mutant coverage* but only for logical expression mutants. Gopinath et al. [68] created higher-order mutants but only used the statement deletion mutation operator. Wedyan and Ghosh [186] and Ferrari et al. [56] investigated mutation testing for *AspectJ* programs. They calculated cost reduction with respect to automatic detection of equivalent mutants of pointcut expressions, which are specific to aspect-oriented programming.

Other examples of varied contexts for mutation testing in which cost reduction has been measured are formal behavioural models [47], and variability points in configurable systems [7].

[10] Aichernig and Jöbstl's 2012 study [5] was subsumed by their 2013 study [6], as shown in Table A.14.

(iv) *Imprecise results*

Some studies reduced costs of mutation testing, but we could not extract precise cost reduction values from the reported results. An example was Kintis et al.'s study [103], which classified 81% of mutants as killable with 71% precision. Even though this was 20% "superior" to previous approaches, we could not define a percentage of cost reduction from the reported data. Schuler et al.'s studies were similar [168, 170].

Kurtz et al. [108] presented several cross-comparison experiments involving redundant, equivalent, and dominator mutants. However, they did not present precise values for cost reduction or mutation scores. Marcozzi et al. [118] addressed the automatic identification of equivalent, redundant, and trivial mutants, which they called *polluting test objectives*. They did not compare their results with baseline or ground-truth numbers of equivalent or trivial mutants.

(v) *Results obtained from inconsistent subjects*

We also found inconsistencies among the sets of subject programs used in some studies. For example, Papadakis and Malevris [150] measured *Mutant execution speedup* based on seven Java programs. In the same study, however, the authors compared results for automatic test generation based on 10 programs that included only five of the original seven programs. Ma and Kim [114] studied cost reduction with *Number of test case executions* and *Mutant execution speedup*. While the cost reduction based on the first metric (*Number of test case executions*) was collected for three groups of subjects, the measurement of results for *Mutant execution speedup* was presented for a single group of programs. As another example, Chen and Zhang [26] measured cost reduction based on four metrics: *Number of test cases required to achieve mutant coverage*; *Number of mutants to be executed*; *Mutant execution speedup*; and *Number of test case executions*. However, the paper presented results only for a subset of subjects for *Mutant execution speedup*.

7. Threats to Validity

This section discusses threats to the validity of our SLR and how we mitigated them. The two main threats to validity are (i) the completeness of the results; and (ii) the classification of studies based on the lists of categories. Regarding the first, the results come from an analysis of 175 peer-reviewed studies published in journals, conference, and workshop proceedings. These studies subsumed other 22 studies. These studies do not include "gray literature," including Masters and PhD theses, technical reports, technical specifications, and other unrefereed documents. This same focus on selecting only peer-reviewed studies was adopted in other SLRs [116, 188]. Furthermore, other surveys on mutation testing (summarized in Section 8) show similar trends, particularly with respect to the growing number of published studies in the last 15 years [82, 117, 144, 173]. The inclusion of additional, non-peer-reviewed, studies would probably not change these results, although it would increase the numbers in this paper.

The backward snowballing technique was used after the first search round with only one level of depth. Wohlin [188] suggested snowballing should be an iterative process that ends only when no new studies are found. Therefore, our procedure to apply snowballing may have created another threat to completeness. To mitigate this threat, we used Wohlin's alternative suggestion for contacting authors of primary studies to ask for additional primary studies. In particular, we contacted every author of selected primary studies to ask for additional papers to be analyzed (see Section 3.3 for more details). As a result, we analyzed 135 more studies suggested in 42 replies, and added 25 items to our final set of studies. The same search procedure was adopted by Jia and Harman [82], who called it a 'transitive closure' on the literature.

For threat (ii), the initial classification of studies based on the *do fewer*, *do smarter*, and *do faster* categories [139] is no longer adequate. When these categories were first defined, fewer techniques were known. More recent techniques are hard to categorize into the *do fewer*, *do smarter*, and *do faster* groups. Category *T-15* for cost reduction techniques reflects this complexity. In addition, since 2000 it has become more common to combine multiple techniques, as can be seen in Figures 6 and 10. As far as possible, the classification we present in this paper was performed in a discerning and unbiased manner.

Still regarding threat (ii), as discussed in Section 5.3, several studies were assigned category *T-15* (*optimization of generation, execution and analysis of mutants*) as an "others" category. It might be possible to further analyze those techniques to identify subcategories, increasing the list of cost reduction techniques. It might also be reasonable to combine some categories, as discussed in Section 5.4.

8. Related Work

In some sense, every paper referenced here is "related." For our related work section, therefore, we focus on other surveys on mutation testing, with a specific emphasis on surveys that focus on cost reduction.

Polo and Reales [159] summarized a small set of reference papers that tried to reduce costs of the main steps in mutation testing: (1) mutant generation, (2) test case generation and execution, and (3) result analysis. The authors suggested techniques such as *T-1* (*random mutation*) and *T-12* (*selective mutation*) for step (1); *T-7* (*minimization and prioritization of test sets*) and *T-3* (*weak mutation*) for step (2); and *T-2* (*higher order mutation*) for step (3). Polo and Reales did not use systematic search criteria to identify the techniques and they did not classify studies by technique. They only analyzed 10 cost reduction-related studies, all of which we include here. One interesting note regarding Polo and Reales's work is that the authors grouped together *test case generation* and *number of test case executions* as a common goal for cost reduction, as we did with our primary goal *PG-4* (Section 5.2).

Jia and Harman [82] published what is probably the most extensive survey of mutation testing in general to date. They identified several cost reduction techniques, including equivalent mutant detection. Jia and Harman identified the first peer-reviewed cost reduction paper as the 1982 paper that introduced *weak mutation* [76]. We include this paper in Figure 6, but not as a selected paper, as it did not focus on cost reduction, did not implement the idea, and included no empirical validation. Our SLR search found the first study that used *weak mutation* to reduce cost to be Marshall et al. [119] in 1990. Jia and Harman summarized 66 mutation testing publications they classified as cost reduction-related, published from 1982 through 2009, including 52 peer-reviewed papers and 14 non-indexed (mostly, gray literature) items such as PhD and Masters Theses, technical reports, and Doctoral Symposium papers. In the same timeframe, our paper includes 44 peer-reviewed studies specifially focused on cost reduction.

Madeyski et al. [117] published results of an SLR on the equivalent mutant problem. The authors classified 24 studies in three categories, based on the goal: *detecting equivalent mutants*, *avoiding equivalent mutant generation*, and *suggesting equivalent mutants*. Moreover, they identified 17 methods that contribute to achieve such goals. This paper includes 15 of Madeyski et al.'s studies; the others either did not focus on cost reduction, were not peer-reviewed, or are included here as related work.

Silva et al. [173] performed an SLR on the use of Search-based Software Testing (SBST) techniques in mutation testing. The authors identified 69 studies published between 1998 and 2014, focusing primarily on the meta-heuristics applied. As opposed to this study, Silva et al. provided an overall characterization of their topic without focusing on cost reduction-related studies. They also included non-peer-reviewed studies.

In the most recent related work, Papadakis et al. [144] updated Jia and Harman's [82] survey to include studies published from 2008 to 2017 (inclusive). The authors stated they selected a total of 502 publications (although only 405 appear in the reference list), of which 260 addressed mutation testing problems (including supporting tools), while 242 either used mutation testing to assess test quality, or tackled problems unrelated to mutation testing. According to Papadakis et al., their study focused on conferences, symposia, workshops, and journals. When contrasted with our work, we searched the same databases that index the conference proceedings and journals they analyzed. Therefore, papers in their dataset that are not in our dataset are probably due to differences in our selection criteria. Papadakis et al.'s paper did not report the selection criteria so we could not compare directly.

9. Conclusion, Recommendations, and Future Work

Mutation testing is a highly investigated and very effective way to generate tests and to assess test quality. However, it is also very expensive. The four major cost factors are (i) a large number of mutants are generated and executed, even for small programs; (ii) test data generation; (iii) test suites are large; and (iv) equivalent mutants. In response, researchers have developed many approaches to reduce the cost of mutation. This paper summarized results of a systematic literature review that characterized the history and the state-of-the-art of cost reduction for mutation testing. We updated and extended our prior paper [54], analyzed the evolution of research on the topic, and summarized its underlying goals, techniques, metrics used, and results achieved.

We analyzed a total of 175 peer-reviewed studies, of which 153 present either original or updated approaches and results for mutation cost reduction. The 21 techniques identified use six main goals to reduce mutation cost. Apart from the *Optimization*-related category (which groups a set of techniques that could not be otherwise classified), the techniques investigated the most are *control-flow analysis*, *evolutionary algorithms*, *selective mutation*, *higher order mutation*, and *data-flow analysis*.

Experimental results were measured with 18 metrics, plus a range of complex, study-specific metrics that were not addressed in this review. The mostly commonly used metrics are the *Number of mutants to be executed*, *Mutant execution speedup*, and the *Number of test cases required to achieve mutant coverage*.

Summary of recommendations: Based on the research questions investigated in this work, we summarize several recommendations.

- *Targeting the primary goals*: Our results reveal that *reducing the number of mutants* (*PG-1*) was the most common primary goal, followed by *executing faster* (*PG-3*), then *automatically detecting equivalent mutants* (*PG-2*). If we look at the last 10 years considered in this research (2009 to 2018), we found 113 studies, 15 of which focused on *PG-2* [31, 47, 56, 75, 97, 99, 100, 103, 118, 126, 142, 145, 154, 168, 170]. Thus, only 13% of the studies from the past decade directly addressed a problem that is widely considered the major obstacle to practical adoption of mutation testing [82, 117]. The total number of studies related to equivalent mutants corroborates this observation. In a recent survey, Papadakis et al. [144] also found that equivalent mutant detection remains an open problem. Therefore, we recommend more research into the problem.

  We found 12 studies (11 in the past decade) that addressed *avoiding the creation of certain mutants* (*PG-5*). Rules to prevent creation of equivalent and redundant mutant can be embedded into mutation operators. Given that this goal impacts the largest number of other goals, including reducing the number of equivalent mutants, we recommend more research on this problem.

- *Applying combined techniques*: We found that mutation cost reduction research has become more interdisciplinary over time. It is true that traditional mutation-specific techniques such as changing the set of operators (*e.g. selective mutation* and *one-op mutation*), *weak mutation*, *higher-order mutation*, and *minimal mutation* are still studied. However, we also found frequent and increasing interest in techniques borrowed from other Computer Science subareas such as software analysis and artificial intelligence.

  We also found several studies that combined techniques to achieve substantial savings. For example, *weak mutation* and *optimization* were applied together to reduce the number of mutants to be executed and mutant execution time, with 97% and 94% savings [202]. As another example, *control-flow analysis* has been combined with *constrained mutation* and *random mutation* in industrial setting [156]. This suggests that stronger collaboration with researchers from different subareas can be very productive.

- *Standardizing and sharing experimental methods, materials and reports*: While collecting cost reduction results from the selected studies, we found lots of differences in experimental design that affected the results. The level of detail in the papers also varied greatly. As a consequence, we could not compare studies with similar goals (*e.g.* to run meta-analysis of results). Moreover, very few experimental materials are available online. Therefore, we recommend the community to create benchmarks that include all software artifacts, including software under test, test cases that reveal actual faults, test cases that kill mutants, mutant programs, mutation execution results in the form of killing matrices, as well as tools to produce and handle such artifacts and directions on their use. This would greatly help researchers design experiments that can be compared and reproduced.

Future work: As with any SLR, it is unlikely that we found all primary studies, as acknowledged as a validity threat in Section 7. Even though several search strategies were used, well-known limitations of scientific writing and repositories mean no search can be perfect. Thus we hope to see future updates to this systematic literature review, performed by either ourselves or other researchers.

Acknowledgements

We would like to thank all 42 authors that suggested studies to be included into the SLR dataset. We also thank George Mason University for hosting Fabiano Ferrari during part of his participation in this research. Finally, we are also thankful for the financial support, detailed as follows: Fabiano Ferrari was supported by the Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) - Brasil, grant #2016/21251-0; and CNPq - Brasil, grant #306310/2016-3; Alessandro Pizzoleto was supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. Jeff Offutt was partly supported by KKS (The Knowledge Foundation), by project 20130085, Testing of Critical System Characteristics (TOCSYC); and partly by a Google educational grant, Self-Paced Learning Increases Retention and Capacity (SPARC).

References

[1] Abuljadayel, A., Wedyan, F., 2018. An Approach for the Generation of Higher Order Mutants Using Genetic Algorithms. International Journal of Intelligent Systems and Applications 10, 34–45.

[2] Acree, A.T., Budd, T.A., DeMillo, R.A., Lipton, R.J., Sayward, F.G., 1979. Mutation Analysis. Technical Report GIT-ICS-79/08. School of Information and Computer Science, Georgia Institute of Technology. Atlanta, GA, USA.

[3] Adamopoulos, K., Harman, M., Hierons, R.M., 2004. How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution, in: Proceedings of the 3rd Genetic and Evolutionary Computation Conference (GECCO), Springer, Seattle, WA, USA. pp. II–1338–1349 (LNCS v.3103).

[4] Aichernig, B.K., Auer, J., Jöbstl, E., Korošec, R., Krenn, W., Schlick, R., Schmidt, B.V., 2014. Model-Based Mutation Testing of an Industrial Measurement Device, in: Proceedings of the 8th International Conference Tests and Proofs (TAP), Springer, York, UK. pp. 1–19.

[5] Aichernig, B.K., Jöbstl, E., 2012. Efficient Refinement Checking for Model-Based Mutation Testing, in: Proceedings of the 12th International Conference on Quality Software (QSIC), IEEE Computer Society, Xi'an, China. pp. 21–30.

[6] Aichernig, B.K., Jöbstl, E., Kegele, M., 2013. Incremental Refinement Checking for Test Case Generation, in: Proceedings of the 7th International Conference Tests and Proofs (TAP), Springer, Budapest, Hungary. pp. 1–19.

[7] Al-Hajjaji, M., Krüger, J., Benduhn, F., Leich, T., Saake, G., 2017. Efficient Mutation Testing in Configurable Systems, in: Proceedings of the IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE), IEEE, Buenos Aires, Argentina. pp. 2–8.

[8] Alexander, R.T., Bieman, J.M., Ghosh, S., Ji, B., 2002. Mutation of Java Objects, in: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE), IEEE, Annapolis, MD, USA. pp. 341–351.

[9] Ammann, P., Delamaro, M.E., Offutt, A.J., 2014. Establishing Theoretical Minimal Sets of Mutants, in: Proceedings of the 7th Conference on Software Testing, Verification and Validation (ICST), IEEE, Cleveland, OH, USA. pp. 21–30.

[10] Anbalagan, P., Xie, T., 2008. Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs, in: Proceeding of the 19th International Symposium on Software Reliability Engineering (ISSRE), IEEE. pp. 239–248.

[11] Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is Mutation an Appropriate Tool for Testing Experiments?, in: Proceedings of the 27th International Conference on Software Engineering (ICSE), ACM, St. Louis, MO, USA. pp. 402–411.

[12] Ayari, K., Bouktif, S., Antoniol, G., 2007. Automatic Mutation Test Input Data Generation via Ant Colony, in: Proceedings of the 9th Genetic and Evolutionary Computation Conference (GECCO), ACM, London, UK. pp. 1074–1081.

[13] Barbosa, E.F., Maldonado, J.C., Vincenzi, A.M.R., 2001. Toward the Determination of Sufficient Mutant Operators for C. Software Testing, Verification and Reliability 11, 113–136.

[14] Bashir, M.B., Nadeem, A., 2017. Improved Genetic Algorithm to Reduce Mutation Testing Cost. IEEE Access 5, 3657–3674.

[15] Baudry, B., Fleurey, F., Jézéquel, J.M., Le Traon, Y., 2005. From Genetic to Bacteriological Algorithms for Mutation-Based Testing. Software Testing, Verification and Reliability 15, 73–96.

[16] Belli, F., Beyazıt, M., 2015. Exploiting Model Morphology for Event-Based Testing. IEEE Transactions on Software Engineering 41, 113–134.

[17] Bieman, J.M., Ghosh, S., Alexander, R.T., 2001. A Technique for Mutation of Java Objects, in: Proceedings of the 16th International Conference on Automated Software Engineering (ASE) - Short Paper, IEEE, San Diego, CA, USA. pp. 337–340.

[18] Biolchini, J., Mian, P.G., Natali, A.C.C., Travassos, G.H., 2005. Systematic Review in Software Engineering. Tech. Report RT-ES 679/05. Systems Engineering and Computer Science Dept., COPPE/UFRJ. Rio de Janeiro, RJ, Brazil.

[19] Bluemke, I., Kulesza, K., 2013. Reduction of Computational Cost in Mutation Testing by Sampling Mutants, in: Proceedings of the 8th International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX), Springer, Brunów, Poland. pp. 41–51.

[20] Bluemke, I., Kulesza, K., 2014a. Reduction in Mutation Testing of Java Classes, in: Proceedings of the 9th International Conference on Software Engineering and Applications (ICSOFT-EA), IEEE, Vienna, Austria. pp. 297–304.

[21] Bluemke, I., Kulesza, K., 2014b. Reductions of Operators in Java Mutation Testing, in: Proceedings of the 9th International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX), Springer, Brunów, Poland. pp. 93–102.

[22] Bogacki, B., Walter, B., 2006a. Aspect-oriented Response Injection: an Alternative to Classical Mutation Testing, in: Proceedings of the IFIP Working Conference on Software Engineering Techniques: Design for Quality (SET), Springer, Warsaw, Poland. pp. 273–282.

[23] Bogacki, B., Walter, B., 2006b. Evaluation of Test Code Quality with Aspect-Oriented Mutations, in: Proceedings of the 7th International Conference Extreme Programming and Agile Processes in Software Engineering (XP) - Posters and Demonstrations, Springer, Oulu, Finland. pp. 202–204.

[24] Budd, T., Angluin, D., 1982. Two Notions of Correctness and their Relation to Testing. Acta

Informatica 8, 31–45.

[25] Cachia, M.A., Micallef, M., Colombo, C., 2013. Towards Incremental Mutation Testing, in: Proceedings of the 2013 Validation Strategies for Software Evolution Workshop (VSSE), Elsevier, Rome, Italy. pp. 2–11.

[26] Chen, L., Zhang, L., 2018. Speeding up Mutation Testing via Regression Test Selection: An Extensive Study, in: Proceedings of the 11th International Conference on Software Testing, Verification and Validation (ICST), IEEE, Västerås, Sweden. pp. 58–69.

[27] Choi, B., Mathur, A.P., 1993. High-Performance Mutation Testing. Journal of Systems and Software 20, 135 – 152.

[28] Delamaro, M.E., Deng, L., Durelli, V.H.S., Li, N., Offutt, A.J., 2014a. Experimental Evaluation of SDL and One-Op Mutation for C, in: Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST), IEEE, Cleveland, OH, USA. pp. 203–212.

[29] Delamaro, M.E., Deng, L., Li, N., Durelli, V.H.S., Offutt, A.J., 2014b. Growing a Reduced Set of Mutation Operators, in: Proceedings of the 28th Brazilian Symposium on Software Engineering (SBES), IEEE, Maceió, AL, Brazil. pp. 81–90.

[30] Delamaro, M.E., Offutt, A.J., Ammann, P., 2014c. Designing Deletion Mutation Operators, in: Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST), IEEE, Cleveland, OH, USA. pp. 11–20.

[31] Delgado-Pérez, P., Medina-Bulo, I., Merayo, M.G., 2017. Using Evolutionary Computation to Improve Mutation Testing, in: Proceedings of the 14th International Work-Conference on Artificial Neural Networks (IWANN), Springer, Cadiz, Spain. pp. 381–391.

[32] Delgado-Pérez, P., Medina-Bulo, I., Palomo-Lozano, F., García-Domínguez, A., Domínguez-Jiménez, J.J., 2017a. Assessment of Class Mutation Operators for C++ with the MuCPP Mutation System. Information and Software Technology , 169–184.

[33] Delgado-Pérez, P., Medina-Bulo, I., Segura, S., García-Domínguez, A., Juan, J., 2017b. GiGAn: Evolutionary Mutation Testing for C++ Object-oriented Systems, in: Proceedings of the 32nd ACM Symposium on Applied Computing (SAC), ACM, Marrakech, Morocco. pp. 1387–1392.

[34] Delgado-Pérez, P., Segura, S., Medina-Bulo, I., 2017c. Assessment of C++ Object-Oriented Mutation Operators: A Selective Mutation Approach. Software Testing, Verification and Reliability 27, 1630 – 1649.

[35] DeMillo, R.A., Krauser, E.W., Mathur, A.P., 1991. Compiler-integrated Program Mutation, in: Proceedings of the 15th IEEE Annual Computer Software and Applications Conference (COMPSAC), IEEE, Tokyo, Japan. pp. 351–356.

[36] DeMillo, R.A., Lipton, R.J., Sayward, F.G., 1978. Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer 11, 34–43.

[37] DeMillo, R.A., Offutt, A.J., 1991. Constraint-based Automatic Test Data Generation. IEEE Transactions on Software Engineering 17, 900–910.

[38] DeMillo, R.A., Offutt, A.J., 1993. Experimental Results from an Automatic Test Case Generator. ACM Transactions on Software Engineering and Methodology 2, 109–127.

[39] Deng, L., Offutt, A.J., Li, N., 2013. Empirical Evaluation of the Statement Deletion Mutation Operator, in: Proceedings of the 6th International Conference on Software Testing, Verification and Validation (ICST), IEEE, Luxembourg City, Luxembourg. pp. 84–93.

[40] Denisov, A., Pankevich, S., 2018. Mull It Over: Mutation Testing Based on LLVM, in: Proceedings of the 13th International Workshop on Mutation Analysis (Mutation), IEEE, Västerås, Sweden. pp. 25–31.

[41] Derezińska, A., 2013. A Quality Estimation of Mutation Clustering in C# Programs, in: Proceedings of the 8th International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX), Springer, Brunów, Poland. pp. 119–129 (AISC v.224).

[42] Derezińska, A., 2016. Evaluation of Deletion Mutation Operators in Mutation Testing of C# Programs, in: Proceedings of the 11th International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX), Springer, Brunów, Poland. pp. 97–108.

[43] Derezińska, A., Hałas, K., 2015. Improving Mutation Testing Process of Python Programs, in: Proceedings of the 4th Computer Science On-line Conference Software Engineering in Intelligent Systems (CSOC), Springer. pp. 233–242.

[44] Derezińska, A., Rudnik, M., 2012. Quality Evaluation of Object-Oriented and Standard Mutation Operators Applied to C# Programs, in: Proceedings of the 50th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS), Springer, Prague, Czech Republic. pp. 42–57.

[45] Derezińska, A., Rudnik, M., 2017. Evaluation of Mutant Sampling Criteria in Object-Oriented Mutation Testing, in: Proceedings of the Federated Conference on Computer Science and Information Systems (ACSIS), Polskie Towarzystwo Informatyczne, Prague, Czech Republic. pp. 1315–1324.

[46] Devroey, X., Perrouin, G., Papadakis, M., Legay, A., Schobbens, P.Y., Heymans, P., 2016. Featured Model-based Mutation Analysis, in: Proceedings of the 38th International Conference on Software Engineering (ICSE), ACM, Austin, TX, USA. pp. 655–666.

[47] Devroey, X., Perrouin, G., Papadakis, M., Legay, A., Schobbens, P.Y., Heymans, P., 2017. Automata Language Equivalence vs. Simulations for Model-Based Mutant Equivalence: An Empirical Evaluation, in: Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, Tokyo, Japan. pp. 424–429.

[48] Do, H., Rothermel, G., 2006. On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques. IEEE Transactions on Software Engineering 32, 733–752.

[49] Domínguez-Jiménez, J.J., Estero-Botaro, A., García-Domínguez, A., Medina-Bulo, I., 2009a. GAmera: An Automatic Mutant Generation System for WS-BPEL Compositions, in: Proceedings of the 7th IEEE European Conference on Web Services (ECOWS), IEEE, Eindhoven, The Netherlands. pp. 97–106.

[50] Domínguez-Jiménez, J.J., Estero-Botaro, A., García-Domínguez, A., Medina-Bulo, I., 2011. Evolutionary Mutation Testing. Information and Software Technology 53, 1108–1123.

[51] Domínguez-Jiménez, J.J., Estero-Botaro, A., Medina-Bulo, I., 2009b. A Framework for Mutant Genetic Generation for WS-BPEL, in: Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), IEEE, Spindler Mlyn, Czech Republic. pp. 229–240.

[52] Durelli, V.H.S., Offutt, A.J., Delamaro, M.E., 2012. Toward Harnessing High-Level Language Virtual Machines for Further Speeding Up Weak Mutation Testing, in: Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST), IEEE, Montreal, QC, Canada. pp. 681–690.

[53] Fernandes, L., Ribeiro, M., Carvalho, L., Gheyi, R., Mongiovi, M., Santos, A., Cavalcanti, A., Ferrari, F.C., Maldonado, J.C., 2017. Avoiding Useless Mutants, in: Proceedings of the 16th ACM International Conference on Generative Programming: Concepts and Experiences (GPCE), ACM, Vancouver, BC, Canada. pp. 187–198.

[54] Ferrari, F.C., Pizzoleto, A.V., Offutt, A.J., 2018a. A Systematic Review of Cost Reduction Techniques for Mutation Testing: Preliminary Results, in: Proceedings of the 13th International Workshop on Mutation Analysis (Mutation), IEEE, Västerås, Sweden. pp. 1–10.

[55] Ferrari, F.C., Pizzoleto, A.V., Offutt, A.J., Fernandes, L., Ribeiro, M., 2018b. SLR on Cost Reduction for Mutation Testing – Companion Website. Online – accessed on July, 2019. http://goo.gl/edyF1n.

[56] Ferrari, F.C., Rashid, A., Maldonado, J.C., 2013. Towards the Practical Mutation Testing of AspectJ Programs. Science of Computer Programming 78, 1639–1662.

[57] Fleyshgakker, V.N., Weiss, S.N., 1994. Efficient Mutation Analysis: A New Approach, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), ACM, Seattle, WA, USA. pp. 185–195.

[58] Frankl, P.G., Weiss, S.N., Hu, C., 1997. All-uses vs Mutation Testing: An Experimental Comparison of Effectiveness. Journal of Systems and Software 38, 235–253.

[59] Fraser, G., Arcuri, A., 2015. Achieving Scalable Mutation-based Generation of Whole Test Suites. Empirical Software Engineering 20, 783–812.

[60] Fraser, G., Zeller, A., 2010. Mutation-Driven Generation of Oracles and Unit Tests, in: Proceedings

of the 19th International Symposium on Software Testing and Analysis (ISSTA), ACM, Trento, Italy. pp. 147–158.

[61] Fraser, G., Zeller, A., 2012. Mutation-Driven Generation of Oracles and Unit Tests. IEEE Transactions on Software Engineering 38, 278–292.

[62] Gligoric, M., Jagannath, V., Luo, Q., Marinov, D., 2012. Efficient Mutation Testing of Multithreaded Code. Software Testing, Verification and Reliability 23, 375–403.

[63] Gligoric, M., Jagannath, V., Marinov, D., 2010. MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code, in: Proceedings of the 3th International Conference on Software Testing, Verification and Validation (ICST), IEEE, Paris, France. pp. 55–64.

[64] Gligoric, M., Zhang, L., Pereira, C., Pokam, G., 2013. Selective Mutation Testing for Concurrent Code, in: Proceedings of the 22nd International Symposium on Software Testing and Analysis (ISSTA), ACM, Lugano, Switzerland. pp. 224–234.

[65] Gopinath, R., Ahmed, I., Alipour, M.A., Jensen, C., Groce, A., 2017. Mutation Reduction Strategies Considered Harmful. IEEE Transactions on Software Reliability 66, 854–874.

[66] Gopinath, R., Alipour, M.A., Ahmed, I., Jensen, C., Groce, A., 2016a. On the Limits of Mutation Reduction Strategies, in: Proceedings of the 38th International Conference on Software Engineering (ICSE), ACM, Austin, TX, USA. pp. 511–522.

[67] Gopinath, R., Jensen, C., Groce, A., 2016b. Topsy-Turvy: A Smarter and Faster Parallelization of Mutation Analysis, in: Proceedings of the 38th International Conference on Software Engineering Companion (ICSE), ACM, Austin, TX, USA. pp. 740–743.

[68] Gopinath, R., Mathis, B., Zeller, A., 2018. If You Can't Kill a Supermutant, You Have a Problem, in: Proceedings of the 13th International Workshop on Mutation Analysis (Mutation), IEEE, Västerås, Sweden. pp. 18–24.

[69] Hamlet, R.G., 1977. Testing Programs with the Aid of a Compiler. IEEE Transactions on Software Engineering 3, 279–290.

[70] Harman, M., Hierons, R.M., Danicic, S., 2000. The Relationship Between Program Dependence and Mutation Analysis, in: Proceedings of the Mutation 2000 Symposium, Kluwer Academic Publishers, San Jose, CA, USA. pp. 5–13.

[71] Harman, M., Jia, Y., Reales, P., Polo, M., 2014. Angels and Monsters: An Empirical Investigation of Potential Test Effectiveness and Efficiency Improvement from Strongly Subsuming Higher Order Mutation, in: Proceedings of the 29th International Conference on Automated Software Engineering (ASE), ACM, Vasteras, Sweden. pp. 397–408.

[72] Henard, C., Papadakis, M., Le Traon, Y., 2014. Mutation-Based Generation of Software Product Line Test Configurations, in: Proceedings of the 6th International Symposium Search-Based Software Engineering (SSBSE), Springer, Fortaleza, CE, Brazil. pp. 92–106.

[73] Hernandes, E.C.M., Zamboni, A.B., Fabbri, S.C.P.F., Di Thommazo, A., 2012. Using GQM and TAM to Evaluate StArt - A Tool that Supports Systematic Review. CLEI Electronic Journal 15, 13–25.

[74] Hierons, R.M., Harman, M., Danicic, S., 1999. Using Program Slicing to Assist in the Detection of Equivalent Mutants. Software Testing, Verification and Reliability 9, 233–262.

[75] Holling, D., Banescu, S., Probst, M., Petrovska, A., Pretschner, A., 2016. Nequivack: Assessing Mutation Score Confidence, in: Proceedings of the 11th International Workshop on Mutation Analysis (Mutation), IEEE, Chicago, IL, USA. pp. 152–161.

[76] Howden, W.E., 1982. Weak Mutation Testing and Completeness of Test Sets. IEEE Transactions on Software Engineering 8, 371–379.

[77] Hu, J., Li, N., Offutt, A.J., 2011. An Analysis of OO Mutation Operators, in: Proceedings of the 6th International Workshop on Mutation Analysis (Mutation), IEEE, Berlin, Germany. pp. 334–341.

[78] Iida, C., Takada, S., 2017. Reducing Mutants with Mutant Killable Precondition, in: Proceedings of the 12th International Workshop on Mutation Analysis (Mutation), IEEE, Tokyo, Japan. pp. 128–133.

[79] Inozemtseva, L., Hemmati, H., Holmes, R., 2013. Using Fault History to Improve Mutation Reduction, in: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering: New Ideas Track (ESEC/FSE), ACM, Saint Petersburg, Russia. pp. 639–642.

[80] Jackson, D., Woodward, M.R., 2000. Parallel Firm Mutation of Java Programs, in: Proceedings of the Mutation 2000 Symposium, Kluwer Academic Publishers, San Jose, CA, USA. pp. 55–61.
[81] Ji, C., Chen, Z., Xu, B., Zhao, Z., 2009. A Novel Method of Mutation Clustering Based on Domain Analysis, in: Proceedings of the 21th International Conference on Software Engineering and Knowledge Engineering (SEKE), Knowledge Systems Institute Graduate School, Boston, MA, USA. pp. 1–6.
[82] Jia, Y., Harman, M., 2011. An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering 37, 649–678.
[83] Just, R., 2014. The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java, in: Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA), ACM, San Jose, CA, USA. pp. 433–436.
[84] Just, R., Ernst, M.D., Fraser, G., 2014a. Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States, in: Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA), ACM, San Jose, CA, USA. pp. 315–326.
[85] Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G., 2014b. Are Mutants a Valid Substitute for Real Faults in Software Testing?, in: Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE), ACM, Hong Kong, China. pp. 654–665.
[86] Just, R., Kapfhammer, G.M., Schweiggert, F., 2011. Using Conditional Mutation to Increase the Efficiency of Mutation Analysis, in: Proceedings of the 6th International Workshop on Automation of Software Test (AST), ACM, Waikiki, Honolulu, HI, USA. pp. 50–56.
[87] Just, R., Kapfhammer, G.M., Schweiggert, F., 2012a. Do Redundant Mutants Affect the Effectiveness and Efficiency of Mutation Analysis?, in: Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, Montreal, QC, Canada. pp. 720–725.
[88] Just, R., Kapfhammer, G.M., Schweiggert, F., 2012b. Using Non-Redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis, in: Proceedings of the IEEE 23th International Symposium on Software Reliability Engineering (ISSRE), IEEE, Dallas, TX, USA. pp. 11–20.
[89] Just, R., Kurtz, B., Ammann, P., 2017. Inferring Mutant Utility from Program Context, in: Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA), ACM, Santa Barbara, CA, USA. pp. 284–294.
[90] Just, R., Schweiggert, F., 2015. Higher Accuracy and Lower Run Time: Efficient Mutation Analysis Using Non-redundant Mutation Operators. Software Testing, Verification and Reliability 25, 490–507.
[91] Kaminski, G., Ammann, P., 2009. Using a Fault Hierarchy to Improve the Efficiency of DNF Logic Mutation Testing, in: Proceedings of the 2nd International Conference on Software Testing, Verification and Validation (ICST), IEEE, Denver, CO, USA. pp. 386–395.
[92] Kaminski, G., Ammann, P., 2011. Reducing Logic Test Set Size while Preserving Fault Detection. Software Testing, Verification and Reliability 21, 155–193.
[93] Kaminski, G., Ammann, P., Offutt, A.J., 2011a. Better Predicate Testing, in: Proceedings of the 6th International Workshop on Automation of Software Test (AST), ACM, Honolulu, HI, USA. pp. 57–63.
[94] Kaminski, G., Ammann, P., Offutt, A.J., 2013. Improving logic-based testing. Journal of Systems and Software 86, 2002 – 2012.
[95] Kaminski, G., Praphamontripong, U., Ammann, P., Offutt, A.J., 2011b. A Logic Mutation Approach to Selective Mutation for Programs and Queries. Information and Software Technology 53, 1137–1152.
[96] Kim, S.W., Ma, Y.S., Kwon, Y.R., 2012. Combining Weak and Strong Mutation for a Noninterpretive Java Mutation System. Software Testing, Verification and Reliability 23, 647–668.
[97] Kintis, M., Malevris, N., 2013. Identifying More Equivalent Mutants via Code Similarity, in: Proceedings of the 20th Asia-Pacific Software Engineering Conference (APSEC), IEEE, Bangkok, Thailand. pp. 180–188.
[98] Kintis, M., Malevris, N., 2014. Using Data Flow Patterns for Equivalent Mutant Detection, in: Proceedings of the 9th International Workshop on Mutation Analysis (Mutation), IEEE, Cleveland, OH, USA. pp. 196–205.
[99] Kintis, M., Malevris, N., 2015. MEDIC: A Static Analysis Framework for Equivalent Mutant Identi-

fication. Information and Software Technology 68, 1–17.

[100] Kintis, M., Papadakis, M., Jia, Y., Malevris, N., Le Traon, Y., Harman, M., 2017. Detecting Trivial Mutant Equivalences via Compiler Optimisations. IEEE Transactions on Software Engineering 44, 1–25.

[101] Kintis, M., Papadakis, M., Malevris, N., 2010. Evaluating Mutation Testing Alternatives: A Collateral Experiment, in: Proceedings of the 17th Asia-Pacific Software Engineering Conference (APSEC), IEEE, Sydney, Australia. pp. 300–309.

[102] Kintis, M., Papadakis, M., Malevris, N., 2012. Isolating First Order Equivalent Mutants via Second Order Mutation, in: Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, Montreal, QC, Canada. pp. 701–710.

[103] Kintis, M., Papadakis, M., Malevris, N., 2015. Employing Second-order Mutation for Isolating First-order Equivalent Mutants. Software Testing, Verification and Reliability 25, 508–535.

[104] Kitchenham, B., 2004. Procedures for Performing Systematic Reviews. Joint Technical Report TR/SE-0401 (Keele) - 0400011T.1 (NICTA). Software Engineering Group - Department of Computer Science - Keele University; and Empirical Software Engineering - National ICT Australia Ltd. Staffordshire, UK; and Eversleigh, Australia.

[105] Kitchenham, B.A., Dybå, T., Jørgensen, M., 2004. Evidence-Based Software Engineering, in: Proceedings of the 26th International Conference on Software Engineering (ICSE), IEEE, Edinburgh, Scotland. pp. 273–281.

[106] Krauser, E.W., Mathur, A.P., Rego, V.J., 1988. High Performance Testing on SIMD Machines, in: Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis, IEEE, Banff, AB, Canada. pp. 171–177.

[107] Krauser, E.W., Mathur, A.P., Rego, V.J., 1991. High Performance Software Testing on SIMD Machines. IEEE Transactions on Software Engineering 17, 403–423.

[108] Kurtz, B., Ammann, P., Offutt, A.J., Delamaro, M.E., Kurtz, M., Gökçe, N., 2016. Analyzing the Validity of Selective Mutation with Dominator Mutants, in: Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE), ACM, Seattle, WA, USA. pp. 571–582.

[109] Lacerda, J.T.S., Ferrari, F.C., 2014. Towards the Establishment of a Sufficient Set of Mutation Operators for AspectJ Programs, in: Proceedings of the 8th Brazilian Workshop on Systematic and Automated Software Testing (SAST), Brazilian Computer Society, Maceio, AL, Brazil. pp. 21–30.

[110] Li, N., Praphamontripong, U., Offutt, A.J., 2009. An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage, in: Proceedings of the 4th International Workshop on Mutation Analysis (Mutation), IEEE, Denver,CO, USA. pp. 220–229.

[111] Li, N., West, M., Escalona, A., Durelli, V.H.S., 2015. Mutation Testing in Practice Using Ruby, in: Proceedings of the 10th International Workshop on Mutation Analysis (Mutation), IEEE, Graz, Austria. pp. 1–6.

[112] Lima, J.A.P., Guizzo, G., Vergilio, S.R., Silva, A.P.C., Filho, H.L.J., Ehrenfried, H.V., 2016. Evaluating Different Strategies for Reduction of Mutation Testing Costs, in: Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing (SAST), ACM, Maringa, PR, Brazil. pp. 4:1–4:10.

[113] Liu, M.H., Gao, Y.F., Shan, J.H., Liu, J.H., Zhang, L., Sun, J.S., 2006. An Approach to Test Data Generation for Killing Multiple Mutants, in: Proceedings of the 22th International Conference on Software Maintenance (ICSM), IEEE, Philadelphia, PA, USA. pp. 113–122.

[114] Ma, Y.S., Kim, S.W., 2016. Mutation Testing Cost Reduction by Clustering Overlapped Mutants. Journal of Systems and Software 115, 18–30.

[115] Ma, Y.S., Offutt, A.J., Kwon, Y.R., 2005. MuJava: An Automated Class Mutation System. Software Testing, Verification and Reliability 15, 97–133.

[116] MacDonell, S., Shepperd, M., Kitchenham, B., Mendes, E., 2010. How Reliable Are Systematic Reviews in Empirical Software Engineering? IEEE Transactions on Software Engineering 36, 676–687.

[117] Madeyski, L., Orzeszyna, W., Torkar, R., Józala, M., 2014. Overcoming the Equivalent Mutant

Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. IEEE Transactions on Software Engineering 40, 23–42.

[118] Marcozzi, M., Bardin, S., Kosmatov, N., Papadakis, M., Prevosto, V., Correnson, L., 2018. Time to Clean Your Test Objectives, in: Proceedings of the 40th International Conference on Software Engineering (ICSE), IEEE, Gothenburg, Sweden. pp. 456–467.

[119] Marshall, A.C., Hedley, D., Riddell, I.J., Hennell, M.A., 1990. Static Dataflow-aided Weak Mutation Analysis (SDAWM). Information and Software Technology 32, 99–104.

[120] Mathur, A.P., 1991. Performance, Effectiveness, and Reliability Issues in Software Testing, in: Proceedings of the 15th IEEE Annual Computer Software and Applications Conference (COMPSAC), IEEE, Tokyo, Japan. pp. 604–605.

[121] Mathur, A.P., 2007. Foundations of Software Testing. 1st ed., Addison-Wesley Professional, Toronto, ON, Canada.

[122] Mathur, A.P., Wong, W.E., 1993. Evaluation of the Cost of Alternative Mutation Testing Strategies, in: Proceedings of the 7th Brazilian Symposium on Software Engineering (SBES), Brazilian Computer Society, João Pessoa, PB, Brazil. pp. 320–335.

[123] Mathur, A.P., Wong, W.E., 1994. An Empirical Comparison of Data Flow and Mutation Based Test Adequacy Criteria. Software Testing, Verification and Reliability 4, 9–31.

[124] Matnei Filho, R.A., Vergilio, S.R., 2015. A Mutation and Multi-objective Test Data Generation Approach for Feature Testing of Software Product Lines, in: Proceedings of the 29th Brazilian Symposium on Software Engineering (SBES), IEEE, Belo Horizonte, MG, Brazil. pp. 21–30.

[125] McMinn, P., Kapfhammer, G.M., Wright, C.J., 2016. Virtual Mutation Analysis of Relational Database Schemas, in: Proceedings of the 11th International Workshop on Automation of Software Test (AST), ACM, Austin, TX, USA. pp. 36–42.

[126] McMinn, P., Wright, C.J., McCurdy, C.J., Kapfhammer, G., 2018. Automatic Detection and Removal of Ineffective Mutants for the Mutation Analysis of Relational Database Schemas (in press). IEEE Transactions on Software Engineering , 1–1.

[127] Mresa, E.S., Bottaci, L., 1999. Efficiency of Mutation Operators and Selective Mutation Strategies: an Empirical Study. Software Testing, Verification and Reliability 9, 205–232.

[128] Nobre, T., Vergilio, S.R., Pozo, A., 2012. Reducing Interface Mutation Costs with Multiobjective Optimization Algorithms. International Journal of Natural Computing Research , 21–40.

[129] Offutt, A.J., Craft, W.M., 1994. Using Compiler Optimization Techniques to Detect Equivalent Mutants. Software Testing, Verification and Reliability 4, 131–154.

[130] Offutt, A.J., Jin, Z., Pan, J., 1999. The Dynamic Domain Reduction Approach to Test Data Generation. Software Practice and Experience 29, 167–193.

[131] Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C., 1996. An Experimental Determination of Sufficient Mutant Operators. ACM Transactions on Software Engineering and Methodology 5, 99–118.

[132] Offutt, A.J., Lee, S.D., 1991. How Strong is Weak Mutation?, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), ACM, Victoria, BC, Canada. pp. 200–213.

[133] Offutt, A.J., Lee, S.D., 1994. An Empirical Evaluation of Weak Mutation. IEEE Transactions on Software Engineering 20, 337–344.

[134] Offutt, A.J., Ma, Y.S., Kwon, Y.R., 2006. The Class-Level Mutants of MuJava, in: Proceedings of the International Workshop on Automation of Software Test (AST), ACM, Shanghai, China. pp. 78–84.

[135] Offutt, A.J., Pan, J., 1996. Detecting Equivalent Mutants and the Feasible Path Problem, in: Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS), ACM, Gaithersburg, MD, USA. pp. 224–236.

[136] Offutt, A.J., Pan, J., 1997. Automatically Detecting Equivalent Mutants and Infeasible Paths. Software Testing, Verification and Reliability 7, 165–192.

[137] Offutt, A.J., Pargas, R.P., Fichter, S.V., Khambekar, P.K., 1992. Mutation Testing of Software Using a MIMD Computer, in: Proceedings of the International Conference on Parallel Processing (ICPP), John Wiley & Sons, Chicago, Illinois, USA. pp. II–257–266.

[138] Offutt, A.J., Rothermel, G., Zapf, C., 1993. An Experimental Evaluation of Selective Mutation, in:

Proceedings of the 15th International Conference on Software Engineering (ICSE), IEEE, Baltimore, MD, USA. pp. 100–107.

[139] Offutt, A.J., Untch, R.H., 2000. Mutation 2000: Uniting the Orthogonal, in: Proceedings of the Mutation 2000 Symposium, Kluwer Academic Publishers, San Jose, CA, USA. pp. 34–44.

[140] Oliveira, A.A.L., Camilo-Junior, C.G., Vincenzi, A.M.R., 2013. A Coevolutionary Algorithm to Automatic Test Case Selection and Mutant in Mutation Testing, in: Proceedings of the 2013 IEEE Congress on Evolutionary Computation (CEC), IEEE, Cancun, Mexico. pp. 829–836.

[141] Omar, E., Ghosh, S., 2012. An Exploratory Study of Higher Order Mutation Testing in Aspect-Oriented Programming, in: Proceedings of the 23th International Symposium on Software Reliability Engineering (ISSRE), IEEE, Dallas, TX, USA. pp. 1–10.

[142] Papadakis, M., Delamaro, M., Le Traon, Y., 2014. Mitigating the Effects of Equivalent Mutants with Mutant Classification Strategies. Science of Computer Programming , 298–319.

[143] Papadakis, M., Jia, Y., Harman, M., Le Traon, Y., 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique, in: Proceedings of the 37th International Conference on Software Engineering (ICSE), ACM, Florence, Italy. pp. 936–946.

[144] Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M., 2019. Mutation Testing Advances: An Analysis and Survey, in: Memon, A.M. (Ed.), Advances in Computers. Elsevier, Amsterdam, The Netherlands. volume 112, pp. 275–378.

[145] Papadakis, M., Le Traon, Y., 2013. Mutation Testing Strategies using Mutant Classification, in: Proceedings of the 28th ACM Symposium on Applied Computing (SAC), ACM, Coimbra, Portugal. pp. 1223–1229.

[146] Papadakis, M., Malevris, N., 2009. An Effective Path Selection Strategy for Mutation Testing, in: Proceedings of the 16th Asia-Pacific Software Engineering Conference (APSEC), IEEE, Batu Ferringhi, Penang, Malaysia. pp. 422–429.

[147] Papadakis, M., Malevris, N., 2010a. An Empirical Evaluation of the First and Second Order Mutation Testing Strategies, in: Proceedings of the 5th International Workshop on Mutation Analysis (Mutation), IEEE, Paris, France. pp. 90–99.

[148] Papadakis, M., Malevris, N., 2010b. Automatic Mutation Test Case Generation via Dynamic Symbolic Execution, in: Proceedings of the IEEE 21th International Symposium on Software Reliability Engineering (ISSRE), IEEE, San Jose, CA, USA. pp. 121–130.

[149] Papadakis, M., Malevris, N., 2011a. Automatic Mutation based Test Data Generation, in: Proceedings of the 13th Genetic and Evolutionary Computation Conference (GECCO): Search-Based Software Engineering Track (Poster Session), ACM, Dublin, Ireland. pp. 247–248.

[150] Papadakis, M., Malevris, N., 2011b. Automatically Performing Weak Mutation with the Aid of Symbolic Execution, Concolic Testing and Search-based Testing. Software Quality Journal 19, 691–723.

[151] Papadakis, M., Malevris, N., 2012. Mutation Based Test Case Generation Via a Path Selection Strategy. Information and Software Technology 54, 915–932.

[152] Papadakis, M., Malevris, N., Kallia, M., 2010. Towards Automating the Generation of Mutation Tests, in: Proceedings of the 5th Workshop on Automation of Software Test (AST), ACM, Cape Town, South Africa. pp. 111–118.

[153] Parsai, A., Murgia, A., Demeyer, S., 2016. Evaluating Random Mutant Selection at Class-level in Projects with Non-adequate Test Suites, in: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE), ACM, Limerick, Ireland. pp. 1–10.

[154] Patel, K., Hierons, R.M., 2016. Resolving the Equivalent Mutant Problem in the Presence of Non-determinism and Coincidental Correctness, in: Proceedings of the 28th International Conference on Testing Software and Systems (ICTSS), Springer, Graz, Austria. pp. 123–138 (LNCS v.9976).

[155] Patrick, M., Oriol, M., Clark, J.A., 2012. MESSI: Mutant Evaluation by Static Semantic Interpretation, in: Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST), IEEE, Montreal, QC, Canada. pp. 711–719.

[156] Petrović, G., Ivanković, M., 2018. State of Mutation Testing at Google, in: Proceedings of the 40th International Conference on Software Engineering - Software Engineering in Practice Track (ICSE-SEIP) (to appear), IEEE, Gothenburg, Sweden. pp. 163–171.

[157] Petrović, G., Ivanković, M., Kurtz, B., Ammann, P., Just, R., 2018. An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions, in: Proceedings of the 13th International Workshop on Mutation Analysis (Mutation), IEEE, Västerås, Sweden. pp. 47–53.

[158] Polo, M., Piattini, M., García-Rodríguez, I., 2009. Decreasing the Cost of Mutation Testing with Second-Order Mutants. Software Testing, Verification and Reliability 19, 111–131.

[159] Polo, M., Reales, P., 2010. Mutation Testing Cost Reduction Techniques: A Survey. IEEE Software 27, 80–86.

[160] Praphamontripong, U., Offutt, A.J., 2017. Finding Redundancy in Web Mutation Operators, in: Proceedings of the 12th International Workshop on Mutation Analysis (Mutation), IEEE, Tokyo, Japan. pp. 134–142.

[161] Quyen, N.T.H., Tung, K.T., Hanh, L.T.M., Binh, N.T., 2016. Improving Mutant Generation for Simulink Models Using Genetic Algorithm, in: Proceedings of the International Conference on Electronics, Information, and Communications (ICEIC), IEEE, Da Nang, Vietnam. pp. 1–4.

[162] Reales, P., Polo, M., 2012. Mutant Execution Cost Reduction: Through MUSIC (Mutant Schema Improved with Extra Code), in: Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST), IEEE, Montreal, QC, Canada. pp. 664–672.

[163] Reales, P., Polo, M., 2013. Parallel Mutation Testing. Software Testing, Verification and Reliability 23, 315–350.

[164] Reales, P., Polo, M., 2014. Reducing Mutation Costs Through Uncovered Mutants. Software Testing, Verification and Reliability 25, 464–489.

[165] Reales, P., Polo, M., Alemán, J.L.F., 2013. Validating Second-Order Mutation at System Level. IEEE Transactions on Software Engineering 39, 570–587.

[166] Reuling, D., Bürdek, J., Rotärmel, S., Lochau, M., Kelter, U., 2015. Fault-based Product-line Testing: Effective Sample Generation Based on Feature-Diagram Mutation, in: Proceedings of the 19th International Conference on Software Product Line (SPLC), ACM, Nashville, TN, USA. pp. 131–140.

[167] Sahinoğlu, M., Spafford, E.H., 1990. A Bayes Sequential Statistical Procedure for Approving Software Products, in: Proceedings of the IFIP Conference on Approving Software Products (ASP), Elsevier, Banff, AB, Canada. pp. 43–56.

[168] Schuler, D., Dallmeier, V., Zeller, A., 2009. Efficient Mutation Testing by Checking Invariant Violations, in: Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA), ACM, Chicago, IL, USA. pp. 69–80.

[169] Schuler, D., Zeller, A., 2010. (Un-)Covering Equivalent Mutants, in: Proceedings of the 3th International Conference on Software Testing, Verification and Validation (ICST), IEEE, Paris, France. pp. 45–54.

[170] Schuler, D., Zeller, A., 2013. Covering and Uncovering Equivalent Mutants. Software Testing, Verification and Reliability 23, 353–374.

[171] Siami-Namin, A., Andrews, J.H., 2006. Finding Sufficient Mutation Operators via Variable Reduction, in: Proceedings of the 2nd Workshop on Mutation Analysis (Mutation), IEEE, Raleigh, NC, USA. pp. 1–10.

[172] Siami-Namin, A., Andrews, J.H., Murdoch, D.J., 2008. Sufficient Mutation Operators for Measuring Test Effectiveness, in: Proceedings of the 30th International Conference on Software Engineering (ICSE), ACM, Leipzig, Germany. pp. 351–360.

[173] Silva, R.A., Souza, S.R.S., Souza, P.S.L., 2017. A Systematic Review on Search Based Mutation Testing. Information and Software Technology 81, 19–35.

[174] Sridharan, M., Siami-Namin, A., 2010. Prioritizing Mutation Operators based on Importance Sampling, in: Proceedings of the IEEE 21th International Symposium on Software Reliability Engineering (ISSRE), IEEE, San Jose, CA, USA. pp. 378–387.

[175] Steimann, F., Thies, A., 2010. From Behaviour Preservation to Behaviour Modification: Constraint-

Based Mutant Generation, in: Proceedings of the 32th International Conference on Software Engineering (ICSE), ACM, Cape Town, South Africa. pp. 425–434.

[176] Sun, C., Pan, L., Wang, Q., Liu, H., Zhang, X., 2017a. An Empirical Study on Mutation Testing of WS-BPEL Programs. Computer Journal 60, 143–158.

[177] Sun, C., Xue, F., Liu, H., Zhang, X., 2017b. A Path-aware Approach to Mutant Reduction in Mutation Testing. Information and Software Technology 81, 65–81.

[178] Tuya, J., Suárez-Cabal, M.J., de la Riva, C., 2007. Mutating database queries. Information and Software Technology 49, 398–417.

[179] Untch, R.H., 1992. Mutation-based Software Testing Using Program Schemata, in: Proceedings of the 30th Annual Southeast Regional Conference (ACM-SE), ACM, Raleigh, NC, USA. pp. 285–291.

[180] Untch, R.H., 2009. On Reduced Neighborhood Mutation Analysis Using a Single Mutagenic Operator, in: Proceedings of the 47th Annual Southeast Regional Conference (ACM-SE), ACM, Clemson, SC, USA. pp. 71–75.

[181] Untch, R.H., Harrold, M.J., Offutt, A.J., 1997. TUMS: Testing Using Mutant Schemata, in: Proceedings of the 35th Annual Southeast Regional Conference (ACM-SE), ACM, Murfreesboro, TN, USA. pp. 174–181.

[182] Untch, R.H., Offutt, A.J., Harrold, M.J., 1993. Mutation Analysis Using Mutant Schemata, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), ACM, Cambridge, MA, USA. pp. 139–148.

[183] Vincenzi, A.M.R., Maldonado, J.C., Barbosa, E.F., Delamaro, M.E., 2001. Unit and Integration Testing Strategies for C Programs Using Mutation. Software Testing, Verification and Reliability 11, 249–268.

[184] Vincenzi, A.M.R., Nakagawa, E.Y., Maldonado, J.C., Delamaro, M.E., Romero, R.A.F., 2002. Bayesian-Learning Based Guidelines to Determine Equivalent Mutants. International Journal of Software Engineering and Knowledge Engineering 12, 675–689.

[185] Wang, B., Xiong, Y., Shi, Y., Zhang, L., Hao, D., 2017. Faster Mutation Analysis via Equivalence Modulo States, in: Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA), ACM, Santa Barbara, CA, USA. pp. 295–306.

[186] Wedyan, F., Ghosh, S., 2012. On Generating Mutants for AspectJ Programs. Information and Software Technology , 900–914.

[187] Weiss, S.N., Fleyshgakker, V.N., 1993. Improved Serial Algorithms for Mutation Analysis, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), ACM, Cambridge, MA, USA. pp. 149–158.

[188] Wohlin, C., 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering, in: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE), ACM, London, UK. pp. 1–10.

[189] Wong, W.E., Maldonado, J.C., Delamaro, M.E., Mathur, A.P., 1994. Constrained Mutation in C Programs, in: Proceedings of the 8th Brazilian Symposium on Software Engineering (SBES), Brazilian Computer Society. pp. 439–452.

[190] Wong, W.E., Mathur, A.P., 1995. Reducing the Cost of Mutation Testing: An Empirical Study. Journal of Systems and Software 31, 185–196.

[191] Wong, W.E., Mathur, A.P., Maldonado, J.C., 1995. Mutation Versus All-uses: An Empirical Evaluation of Cost, Strength and Effectiveness. Software Quality and Productivity: Theory, Practice and Training , 258–265.

[192] Woodward, M.R., Halewood, K., 1988. From Weak to Strong, Dead or Alive? An Analysis of some Mutation Testing Issues, in: Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis, IEEE, Banff, AB, Canada. pp. 152–158.

[193] Wright, C.J., Kapfhammer, G.M., McMinn, P., 2013. Efficient Mutation Analysis of Relational Database Structure Using Mutant Schemata and Parallelisation, in: Proceedings of the 8th International Workshop on Mutation Analysis (Mutation), IEEE, Luxembourg City, Luxembourg. pp. 63–72.

[194] Wright, C.J., Kapfhammer, G.M., McMinn, P., 2014. The Impact of Equivalent, Redundant and Quasi Mutants on Database Schema Mutation Analysis, in: Proceedings of the 14th International Conference on Quality Software (QSIC), IEEE, Dallas, TX, USA. pp. 57–66.

[195] Zhang, J., Wangi, Z., Zhang, L., Hao, D., Zang, L., Cheng, S., Zhang, L., 2016. Predictive Mutation Testing, in: Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA), ACM, Saarbrücken, Germany. pp. 342–353.

[196] Zhang, L., Gligoric, M., Marinov, D., Khurshid, S., 2013a. Operator-Based and Random Mutant Selection: Better Together, in: Proceedings of the 28th International Conference on Automated Software Engineering (ASE), IEEE, Palo Alto, CA, USA. pp. 92–102.

[197] Zhang, L., Hou, S., Hu, J., Xie, T., Mei, H., 2010a. Is Operator-Based Mutant Selection Superior to Random Mutant Selection?, in: Proceedings of the 32th International Conference on Software Engineering (ICSE), ACM, Cape Town, South Africa. pp. 435–444.

[198] Zhang, L., Marinov, D., Khurshid, S., 2013b. Faster Mutation Testing Inspired by Test Prioritization and Reduction, in: Proceedings of the 22nd International Symposium on Software Testing and Analysis (ISSTA), ACM, Lugano, Switzerland. pp. 235–245.

[199] Zhang, L., Marinov, D., Zhang, L., Khurshid, S., 2012. Regression Mutation Testing, in: Proceedings of the 21st International Symposium on Software Testing and Analysis (ISSTA), ACM, Minneapolis, MN, USA. pp. 331–341.

[200] Zhang, L., Xie, T., Zhang, L., Tillmann, N., de Halleux, J., Mei, H., 2010b. Test Generation via Dynamic Symbolic Execution for Mutation Testing, in: Proceedings of the 26th International Conference on Software Maintenance (ICSM), IEEE, Timisoara, Romania. pp. 1–10.

[201] Zhu, Q., Panichella, A., Zaidman, A., 2017. Speeding-Up Mutation Testing via Data Compression and State Infection, in: Proceedings of the 12th International Workshop on Mutation Analysis (Mutation), IEEE, Tokyo, Japan. pp. 103–109.

[202] Zhu, Q., Panichella, A., Zaidman, A., 2018. An Investigation of Compression Techniques to Speed up Mutation Testing, in: Proceedings of the 11th International Conference on Software Testing, Verification and Validation (ICST), IEEE, Västerås, Sweden. pp. 274–284.

Appendix A. Selected Studies and Subsumed Studies

This appendix includes tables that list all primary studies that were selected in this SLR (Tables A.7 to A.13), and a table that lists all subsumed studies (Tables A.14 and A.15). Tables of selected studies are sorted by author name and year of study publication, while tables of subsumed studies are sorted by year of publication of the updated or extended studies.

Table A.7: Selected Studies (1/7) (Id = Reference ID)

Id	Author	Title	Year	Publisher	Journal/Event
[1]	Abuljadayel and Wedyan	An Approach for the Generation of Higher Order Mutants Using Genetic Algorithms	2018	Modern Education and Computer Science Press	International Journal of Intelligent Systems and Applications
[6]	Aichernig, Jöbstl and Kegele	Incremental Refinement Checking for Test Case Generation	2013	Springer	International Conference on Quality Software (QSIC)
[4]	Aichernig, Auer, Jöbstl, Korošec, Krenn, Schlick and Schmidt	Model-Based Mutation Testing of an Industrial Measurement Device	2014	Springer	International Conference Tests and Proofs (TAP)
[7]	Al-Hajjaji, Krüger, Benduhn, Leich and Saake	Efficient Mutation Testing in Configurable Systems	2017	IEEE	International Workshop on Variability and Complexity in Software Design (VACE)
[8]	Alexander, Bieman, Ghosh and Ji	Mutation of Java Objects	2002	IEEE	International Symposium on Software Reliability Engineering (ISSRE)
[9]	Ammann, Delamaro and Offutt	Establishing Theoretical Minimal Sets of Mutants	2014	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
[10]	Anbalagan and Xie	Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs	2008	IEEE	International Symposium on Software Reliability Engineering (ISSRE)
[3]	Adamopoulos, Harman and Hierons	How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution	2004	Springer	Genetic and Evolutionary Computation Conference (GECCO)
[12]	Ayari, Bouktif and Antoniol	Automatic Mutation Test Input Data Generation via Ant Colony	2007	ACM	Genetic and Evolutionary Computation Conference (GECCO)
[13]	Barbosa, Maldonado and Vincenzi	Toward the Determination of Sufficient Mutant Operators for C	2001	Wiley	Software Testing, Verification and Reliability
[14]	Bashir and Nadeem	Improved Genetic Algorithm to Reduce Mutation Testing Cost	2017	IEEE	IEEE Access
[15]	Baudry, Fleurey, Jézéquel and Le Traon	From Genetic to Bacteriological Algorithms for Mutation-Based Testing	2005	Wiley	Software Testing, Verification and Reliability
[16]	Belli and Beyazıt	Exploiting Model Morphology for Event-Based Testing	2015	IEEE	IEEE Transactions on Software Engineering
[19]	Bluemke and Kulesza	Reduction of Computational Cost in Mutation Testing by Sampling Mutants	2013	Springer	International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)
[21]	Bluemke and Kulesza	Reductions of Operators in Java Mutation Testing	2014b	Springer	International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)
[20]	Bluemke and Kulesza	Reduction in Mutation Testing of Java Classes	2014a	IEEE	International Conference on Software Engineering and Applications (ICSOFT-EA)
[22]	Bogacki and Walter	Aspect-oriented Response Injection: an Alternative to Classical Mutation Testing	2006a	Springer	IFIP Working Conference on Software Engineering Techniques: Design for Quality (SET)
[27]	Choi and Mathur	High-Performance Mutation Testing	1993	Elsevier	Journal of Systems and Software
[25]	Cachia, Micallef and Colombo	Towards Incremental Mutation Testing	2013	Elsevier	Validation Strategies for Software Evolution Workshop (VSSE)
[26]	Chen and Zhang	Speeding up Mutation Testing via Regression Test Selection: An Extensive Study	2018	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
[28]	Delamaro, Deng, Durelli, Li and Offutt	Experimental Evaluation of SDL and One-Op Mutation for C	2014a	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
[30]	Delamaro, Offutt and Ammann	Designing Deletion Mutation Operators	2014c	IEEE	International Conference on Software Testing, Verification and Validation (ICST)

Table A.8: Selected Studies (2/7) (Id = Reference ID)

Id	Author	Title	Year	Publisher	Journal/Event
[29]	Delamaro, Deng, Li, Durelli and Offutt	Growing a Reduced Set of Mutation Operators	2014b	IEEE	Brazilian Symposium on Software Engineering (SBES)
[33]	Delgado-Pérez, Medina-Bulo, Segura, García-Domínguez and Juan	GiGAn: Evolutionary Mutation Testing for C++ Object-oriented Systems	2017b	ACM	ACM Symposium on Applied Computing (SAC)
[34]	Delgado-Pérez, Segura and Medina-Bulo	Assessment of C++ Object-Oriented Mutation Operators: A Selective Mutation Approach	2017c	Wiley	Software Testing, Verification and Reliability
[32]	Delgado-Pérez, Medina-Bulo, Palomo-Lozano, García-Domínguez and Domínguez-Jiménez	Assessment of Class Mutation Operators for C++ with the MuCPP Mutation System	2017a	Elsevier	Information and Software Technology
[31]	Delgado-Pérez, Medina-Bulo and Merayo	Using Evolutionary Computation to Improve Mutation Testing	2017	Springer	International Work-Conference on Artificial Neural Networks (IWANN)
[35]	DeMillo, Krauser and Mathur	Compiler-integrated Program Mutation	1991	IEEE	IEEE Annual Computer Software and Applications Conference (COMPSAC)
[37]	DeMillo and Offutt	Constraint-based Automatic Test Data Generation	1991	IEEE	IEEE Transactions on Software Engineering
[38]	DeMillo and Offutt	Experimental Results from an Automatic Test Case Generator	1993	ACM	ACM Transactions on Software Engineering and Methodology
[39]	Deng, Offutt and Li	Empirical Evaluation of the Statement Deletion Mutation Operator	2013	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
[40]	Denisov and Pankevich	Mull It Over: Mutation Testing Based on LLVM	2018	IEEE	International Workshop on Mutation Analysis (Mutation)
[44]	Derezińska and Rudnik	Quality Evaluation of Object-Oriented and Standard Mutation Operators Applied to C# Programs	2012	Springer	International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS)
[41]	Derezińska	A Quality Estimation of Mutation Clustering in C# Programs	2013	Springer	International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)
[43]	Derezińska and Hałas	Improving Mutation Testing Process of Python Programs	2015	Springer	Computer Science On-line Conference Software Engineering in Intelligent Systems (CSOC)
[42]	Derezińska	Evaluation of Deletion Mutation Operators in Mutation Testing of C# Programs	2016	Springer	International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)
[45]	Derezińska and Rudnik	Evaluation of Mutant Sampling Criteria in Object-Oriented Mutation Testing	2017	Polskie Towarzystwo Informatyczne	Federated Conference on Computer Science and Information Systems (ACSIS)
[46]	Devroey, Perrouin, Papadakis, Legay, Schobbens and Heymans	Featured Model-based Mutation Analysis	2016	ACM	International Conference on Software Engineering (ICSE)
[47]	Devroey, Perrouin, Papadakis, Legay, Schobbens and Heymans	Automata Language Equivalence vs. Simulations for Model-Based Mutant Equivalence: An Empirical Evaluation	2017	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
[49]	Domínguez-Jiménez, Estero-Botaro, García-Domínguez and Medina-Bulo	GAmera: An Automatic Mutant Generation System for WS-BPEL Compositions	2009a	IEEE	IEEE European Conference on Web Services (ECOWS)
[50]	Domínguez-Jiménez, Estero-Botaro, García-Domínguez and Medina-Bulo	Evolutionary Mutation Testing	2011	Elsevier	Information and Software Technology
[52]	Durelli, Offutt and Delamaro	Toward Harnessing High-Level Language Virtual Machines for Further Speeding Up Weak Mutation Testing	2012	IEEE	International Conference on Software Testing, Verification and Validation (ICST)

Table A.9: Selected Studies (3/7) (Id = Reference ID)

Id	Author	Title	Year	Publisher	Journal/Event
[53]	Fernandes, Ribeiro, Carvalho, Gheyi, Mongiovi, Santos, Cavalcanti, Ferrari and Maldonado	Avoiding Useless Mutants	2017	ACM	International Conference on Generative Programming: Concepts and Experiences (GPCE)
[56]	Ferrari, Rashid and Maldonado	Towards the Practical Mutation Testing of AspectJ Programs	2013	Elsevier	Science of Computer Programming
[124]	Matnei Filho and Vergilio	A Mutation and Multi-objective Test Data Generation Approach for Feature Testing of Software Product Lines	2015	IEEE	Brazilian Symposium on Software Engineering (SBES)
[57]	Fleyshgakker and Weiss	Efficient Mutation Analysis: A New Approach	1994	ACM	International Symposium on Software Testing and Analysis (ISSTA)
[61]	Fraser and Zeller	Mutation-Driven Generation of Oracles and Unit Tests	2012	IEEE	IEEE Transactions on Software Engineering
[59]	Fraser and Arcuri	Achieving Scalable Mutation-based Generation of Whole Test Suites	2015	Springer	Empirical Software Engineering
[63]	Gligoric, Jagannath and Marinov	MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code	2010	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
[62]	Gligoric, Jagannath, Luo and Marinov	Efficient Mutation Testing of Multithreaded Code	2012	Wiley	Software Testing, Verification and Reliability
[64]	Gligoric, Zhang, Pereira and Pokam	Selective Mutation Testing for Concurrent Code	2013	ACM	International Symposium on Software Testing and Analysis (ISSTA)
[67]	Gopinath, Jensen and Groce	Topsy-Turvy: A Smarter and Faster Parallelization of Mutation Analysis	2016b	ACM	International Conference on Software Engineering (ICSE)
[65]	Gopinath, Ahmed, Alipour, Jensen and Groce	Mutation Reduction Strategies Considered Harmful	2017	IEEE	IEEE Transactions on Software Reliability
[68]	Gopinath, Mathis and Zeller	If You Can't Kill a Supermutant, You Have a Problem	2018	IEEE	International Workshop on Mutation Analysis (Mutation)
[70]	Harman, Hierons and Danicic	The Relationship Between Program Dependence and Mutation Analysis	2000	Kluwer	Mutation 2000 Symposium
[71]	Harman, Jia, Reales and Polo	Angels and Monsters: An Empirical Investigation of Potential Test Effectiveness and Efficiency Improvement from Strongly Subsuming Higher Order Mutation	2014	ACM	International Conference on Automated Software Engineering (ASE)
[74]	Hierons, Harman and Danicic	Using Program Slicing to Assist in the Detection of Equivalent Mutants	1999	Wiley	Software Testing, Verification and Reliability
[72]	Henard, Papadakis and Le Traon	Mutation-Based Generation of Software Product Line Test Configurations	2014	Springer	International Symposium Search-Based Software Engineering (SSBSE)
[75]	Holling, Banescu, Probst, Petrovska and Pretschner	Nequivack: Assessing Mutation Score Confidence	2016	IEEE	International Workshop on Mutation Analysis (Mutation)
[77]	Hu, Li and Offutt	An Analysis of OO Mutation Operators	2011	IEEE	International Workshop on Mutation Analysis (Mutation)
[78]	Iida and Takada	Reducing Mutants with Mutant Killable Precondition	2017	IEEE	International Workshop on Mutation Analysis (Mutation)
[79]	Inozemtseva, Hemmati and Holmes	Using Fault History to Improve Mutation Reduction	2013	ACM	International Symposium on the Foundations of Software Engineering (FSE)
[80]	Jackson and Woodward	Parallel Firm Mutation of Java Programs	2000	Kluwer	Mutation 2000 Symposium
[81]	Ji, Chen, Xu and Zhao	A Novel Method of Mutation Clustering Based on Domain Analysis	2009	Knowledge Systems Institute Graduate School	International Conference on Software Engineering and Knowledge Engineering (SEKE)
[86]	Just, Kapfhammer and Schweiggert	Using Conditional Mutation to Increase the Efficiency of Mutation Analysis	2011	ACM	International Workshop on Automation of Software Test (AST)

Table A.10: Selected Studies (4/7) (Id = Reference ID)

Id	Author	Title	Year	Publisher	Journal/Event
[88]	Just, Kapfhammer and Schweiggert	Using Non-Redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis	2012b	IEEE	International Symposium on Software Reliability Engineering (ISSRE)
[84]	Just, Ernst and Fraser	Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States	2014a	ACM	International Symposium on Software Testing and Analysis (ISSTA)
[90]	Just and Schweiggert	Higher Accuracy and Lower Run Time: Efficient Mutation Analysis Using Non-redundant Mutation Operators	2015	Wiley	Software Testing, Verification and Reliability
[89]	Just, Kurtz and Ammann	Inferring Mutant Utility from Program Context	2017	ACM	International Symposium on Software Testing and Analysis (ISSTA)
[91]	Kaminski and Ammann	Using a Fault Hierarchy to Improve the Efficiency of DNF Logic Mutation Testing	2009	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
[92]	Kaminski and Ammann	Reducing Logic Test Set Size while Preserving Fault Detection	2011	Wiley	Software Testing, Verification and Reliability
[95]	Kaminski, Praphamontripong, Ammann and Offutt	A Logic Mutation Approach to Selective Mutation for Programs and Queries	2011b	Elsevier	Information and Software Technology
[94]	Kaminski, Ammann and Offutt	Improving logic-based testing	2013	Elsevier	Journal of Systems and Software
[96]	Kim, Ma and Kwon	Combining Weak and Strong Mutation for a Noninterpretive Java Mutation System	2012	Wiley	Software Testing, Verification and Reliability
[101]	Kintis, Papadakis and Malevris	Evaluating Mutation Testing Alternatives: A Collateral Experiment	2010	IEEE	Asia-Pacific Software Engineering Conference (APSEC)
[97]	Kintis and Malevris	Identifying More Equivalent Mutants via Code Similarity	2013	IEEE	Asia-Pacific Software Engineering Conference (APSEC)
[98]	Kintis and Malevris	Using Data Flow Patterns for Equivalent Mutant Detection	2014	IEEE	International Workshop on Mutation Analysis (Mutation)
[103]	Kintis, Papadakis and Malevris	Employing Second-order Mutation for Isolating First-order Equivalent Mutants	2015	Wiley	Software Testing, Verification and Reliability
[99]	Kintis and Malevris	MEDIC: A Static Analysis Framework for Equivalent Mutant Identification	2015	Elsevier	Information and Software Technology
[100]	Kintis, Papadakis, Jia, Malevris, Le Traon and Harman	Detecting Trivial Mutant Equivalences via Compiler Optimisations	2017	IEEE	IEEE Transactions on Software Engineering
[107]	Krauser, Mathur and Rego	High Performance Software Testing on SIMD Machines	1991	IEEE	IEEE Transactions on Software Engineering
[108]	Kurtz, Ammann, Offutt, Delamaro, Kurtz and Gökçe	Analyzing the Validity of Selective Mutation with Dominator Mutants	2016	ACM	International Symposium on the Foundations of Software Engineering (FSE)
[109]	Lacerda and Ferrari	Towards the Establishment of a Sufficient Set of Mutation Operators for AspectJ Programs	2014	SBC	Brazilian Symposium on Systematic and Automated Software Testing (SAST)
[111]	Li, West, Escalona and Durelli	Mutation Testing in Practice Using Ruby	2015	IEEE	International Workshop on Mutation Analysis (Mutation)
[112]	Lima, Guizzo, Vergilio, Silva, Filho and Ehrenfried	Evaluating Different Strategies for Reduction of Mutation Testing Costs	2016	ACM	Brazilian Symposium on Systematic and Automated Software Testing (SAST)
[113]	Liu, Gao, Shan, Liu, Zhang and Sun	An Approach to Test Data Generation for Killing Multiple Mutants	2006	IEEE	International Conference on Software Maintenance (ICSM)
[115]	Ma, Offutt and Kwon	MuJava: An Automated Class Mutation System	2005	Wiley	Software Testing, Verification and Reliability
[114]	Ma and Kim	Mutation Testing Cost Reduction by Clustering Overlapped Mutants	2016	Elsevier	Journal of Systems and Software
[117]	Madeyski, Orzeszyna, Torkar and Józala	Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation	2014	IEEE	IEEE Transactions on Software Engineering

Table A.11: Selected Studies (5/7) (Id = Reference ID)

Id	Author	Title	Year	Publisher	Journal/Event
[118]	Marcozzi, Bardin, Kosmatov, Papadakis, Prevosto and Correnson	Time to Clean Your Test Objectives	2018	IEEE	International Conference on Software Engineering (ICSE)
[119]	Marshall, Hedley, Riddell and Hennell	Static Dataflow-aided Weak Mutation Analysis (SDAWM)	1990	Elsevier	Information and Software Technology
[163]	Reales and Polo	Parallel Mutation Testing	2013	wiley	Software Testing, Verification and Reliability
[165]	Reales, Polo and Alemán	Validating Second-Order Mutation at System Level	2013	IEEE	IEEE Transactions on Software Engineering
[164]	Reales and Polo	Reducing Mutation Costs Through Uncovered Mutants	2014	Wiley	Software Testing, Verification and Reliability
[125]	McMinn, Kapfhammer and Wright	Virtual Mutation Analysis of Relational Database Schemas	2016	IEEE	International Workshop on Automation of Software Test (AST)
[126]	McMinn, Wright, McCurdy and Kapfhammer	Automatic Detection and Removal of Ineffective Mutants for the Mutation Analysis of Relational Database Schemas	2018	IEEE	IEEE Transactions on Software Engineering
[127]	Mresa and Bottaci	Efficiency of Mutation Operators and Selective Mutation Strategies: an Empirical Study	1999	Wiley	Software Testing, Verification and Reliability
[171]	Siami-Namin and Andrews	Finding Sufficient Mutation Operators via Variable Reduction	2006	IEEE	International Workshop on Mutation Analysis (Mutation)
[172]	Siami-Namin, Andrews and Murdoch	Sufficient Mutation Operators for Measuring Test Effectiveness	2008	ACM	International Conference on Software Engineering (ICSE)
[128]	Nobre, Vergilio and Pozo	Reducing Interface Mutation Costs with Multiobjective Optimization Algorithms	2012	IGI Global	International Journal of Natural Computing Research
[137]	Offutt, Pargas, Fichter and Khambekar	Mutation Testing of Software Using a MIMD Computer	1992	Wiley	International Conference on Parallel Processing (ICPP)
[138]	Offutt, Rothermel and Zapf	An Experimental Evaluation of Selective Mutation	1993	IEEE	International Conference on Software Engineering (ICSE)
[129]	Offutt and Craft	Using Compiler Optimization Techniques to Detect Equivalent Mutants	1994	Wiley	Software Testing, Verification and Reliability
[133]	Offutt and Lee	An Empirical Evaluation of Weak Mutation	1994	IEEE	IEEE Transactions on Software Engineering
[131]	Offutt, Lee, Rothermel, Untch and Zapf	An Experimental Determination of Sufficient Mutant Operators	1996	ACM	ACM Transactions on Software Engineering and Methodology
[136]	Offutt and Pan	Automatically Detecting Equivalent Mutants and Infeasible Paths	1997	Wiley	Software Testing, Verification and Reliability
[130]	Offutt, Jin and Pan	The Dynamic Domain Reduction Procedure for Test Data Generation	1999	Wiley	Software Practice and Experience
[134]	Offutt, Ma and Kwon	The Class-Level Mutants of MuJava	2006	ACM	International Workshop on Automation of Software Test (AST)
[140]	Oliveira, Camilo-Junior and Vincenzi	A Coevolutionary Algorithm to Automatic Test Case Selection and Mutant in Mutation Testing	2013	IEEE	IEEE Congress on Evolutionary Computation (CEC)
[141]	Omar and Ghosh	An Exploratory Study of Higher Order Mutation Testing in Aspect-Oriented Programming	2012	IEEE	International Symposium on Software Reliability Engineering (ISSRE)
[146]	Papadakis and Malevris	An Effective Path Selection Strategy for Mutation Testing	2009	IEEE	Asia-Pacific Software Engineering Conference (APSEC)
[147]	Papadakis and Malevris	An Empirical Evaluation of the First and Second Order Mutation Testing Strategies	2010a	IEEE	International Workshop on Mutation Analysis (Mutation)
[148]	Papadakis and Malevris	Automatic Mutation Test Case Generation via Dynamic Symbolic Execution	2010b	IEEE	International Symposium on Software Reliability Engineering (ISSRE)
[149]	Papadakis and Malevris	Automatic Mutation based Test Data Generation	2011a	ACM	Genetic and Evolutionary Computation Conference (GECCO)
[150]	Papadakis and Malevris	Automatically Performing Weak Mutation with the Aid of Symbolic Execution, Concolic Testing and Search-based Testing	2011b	Springer	Software Quality Journal

Table A.12: Selected Studies (6/7) (Id = Reference ID)

Id	Author	Title	Year	Publisher	Journal/Event
[151]	Papadakis and Malevris	Mutation Based Test Case Generation Via a Path Selection Strategy	2012	Elsevier	Information and Software Technology
[145]	Papadakis and Le Traon	Mutation Testing Strategies using Mutant Classification	2013	ACM	ACM Symposium on Applied Computing (SAC)
[142]	Papadakis, Delamaro and Le Traon	Mitigating the Effects of Equivalent Mutants with Mutant Classification Strategies	2014	Elsevier	Science of Computer Programming
[153]	Parsai, Murgia and Demeyer	Evaluating Random Mutant Selection at Class-level in Projects with Non-adequate Test Suites	2016	ACM	International Conference on Evaluation and Assessment in Software Engineering (EASE)
[154]	Patel and Hierons	Resolving the Equivalent Mutant Problem in the Presence of Non-determinism and Coincidental Correctness	2016	Springer	International Conference on Testing Software and Systems (ICTSS)
[155]	Patrick, Oriol and Clark	MESSI: Mutant Evaluation by Static Semantic Interpretation	2012	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
[156]	Petrović and Ivanković	State of Mutation Testing at Google	2018	IEEE	International Conference on Software Engineering - Software Engineering in Practice Track (ICSE-SEIP)
[160]	Praphamontripong and Offutt	Finding Redundancy in Web Mutation Operators	2017	IEEE	International Workshop on Mutation Analysis (Mutation)
[161]	Quyen, Tung, Hanh and Binh	Improving Mutant Generation for Simulink Models Using Genetic Algorithm	2016	IEEE	International Conference on Electronics, Information, and Communications (ICEIC)
[166]	Reuling, Bürdek, Rotärmel, Lochau and Kelter	Fault-based Product-line Testing: Effective Sample Generation Based on Feature-Diagram Mutation	2015	ACM	International Conference on Software Product Line (SPLC)
[167]	Sahinoğlu and Spafford	A Bayes Sequential Statistical Procedure for Approving Software Products	1990	Elsevier	IFIP Conference on Approving Software Products (ASP)
[168]	Schuler, Dallmeier and Zeller	Efficient Mutation Testing by Checking Invariant Violations	2009	ACM	International Symposium on Software Testing and Analysis (ISSTA)
[170]	Schuler and Zeller	Covering and Uncovering Equivalent Mutants	2013	Wiley	Software Testing, Verification and Reliability
[174]	Sridharan and Siami-Namin	Prioritizing Mutation Operators based on Importance Sampling	2010	IEEE	International Symposium on Software Reliability Engineering (ISSRE)
[175]	Steimann and Thies	From Behaviour Preservation to Behaviour Modification: Constraint-Based Mutant Generation	2010	ACM	International Conference on Software Engineering (ICSE)
[176]	Sun, Pan, Wang, Liu and Zhang	An Empirical Study on Mutation Testing of WS-BPEL Programs	2017a	Oxford University Press	Computer Journal
[177]	Sun, Xue, Liu and Zhang	A Path-aware Approach to Mutant Reduction in Mutation Testing	2017b	Elsevier	Information and Software Technology
[178]	Tuya, Suárez-Cabal and de la Riva	Mutating database queries	2007	Elsevier	Information and Software Technology
[182]	Untch, Offutt and Harrold	Mutation Analysis Using Mutant Schemata	1993	ACM	International Symposium on Software Testing and Analysis (ISSTA)
[181]	Untch, Harrold and Offutt	TUMS: Testing Using Mutant Schemata	1997	ACM	Annual Southeast Regional Conference (ACM-SE)
[180]	Untch	On Reduced Neighborhood Mutation Analysis Using a Single Mutagenic Operator	2009	ACM	Annual Southeast Regional Conference (ACM-SE)
[158]	Polo, Piattini and García-Rodríguez	Decreasing the Cost of Mutation Testing with Second-Order Mutants	2009	Wiley	Software Testing, Verification and Reliability
[183]	Vincenzi, Maldonado, Barbosa and Delamaro	Unit and Integration Testing Strategies for C Programs Using Mutation	2001	Wiley	Software Testing, Verification and Reliability
[184]	Vincenzi, Nakagawa, Maldonado, Delamaro and Romero	Bayesian-Learning Based Guidelines to Determine Equivalent Mutants	2002	World Scientific Publishing	International Journal of Software Engineering and Knowledge Engineering

Table A.13: Selected Studies (7/7) (Id = Reference ID)

Id	Author	Title	Year	Publisher	Journal/Event
[185]	Wang, Xiong, Shi, Zhang and Hao	Faster Mutation Analysis via Equivalence Modulo States	2017	ACM	International Symposium on Software Testing and Analysis (ISSTA)
[186]	Wedyan and Ghosh	On Generating Mutants for AspectJ Programs	2012	Elsevier	Information and Software Technology
[187]	Weiss and Fleyshgakker	Improved Serial Algorithms for Mutation Analysis	1993	ACM	International Symposium on Software Testing and Analysis (ISSTA)
[189]	Wong, Maldonado, Delamaro and Mathur	Constrained Mutation in C Programs	1994	Brazilian Computer Society	Brazilian Symposium on Software Engineering (SBES)
[190]	Wong and Mathur	Reducing the Cost of Mutation Testing: An Empirical Study	1995	Elsevier	Journal of Systems and Software
[193]	Wright, Kapfhammer and McMinn	Efficient Mutation Analysis of Relational Database Structure Using Mutant Schemata and Parallelisation	2013	IEEE	International Workshop on Mutation Analysis (Mutation)
[197]	Zhang, Hou, Hu, Xie and Mei	Is Operator-Based Mutant Selection Superior to Random Mutant Selection?	2010a	ACM	International Conference on Software Engineering (ICSE)
[200]	Zhang, Xie, Zhang, Tillmann, de Halleux and Mei	Test Generation via Dynamic Symbolic Execution for Mutation Testing	2010b	IEEE	International Conference on Software Maintenance (ICSM)
[199]	Zhang, Marinov, Zhang and Khurshid	Regression Mutation Testing	2012	ACM	International Symposium on Software Testing and Analysis (ISSTA)
[198]	Zhang, Marinov and Khurshid	Faster Mutation Testing Inspired by Test Prioritization and Reduction	2013b	ACM	International Symposium on Software Testing and Analysis (ISSTA)
[196]	Zhang, Gligoric, Marinov and Khurshid	Operator-Based and Random Mutant Selection: Better Together	2013a	IEEE	International Conference on Automated Software Engineering (ASE)
[195]	Zhang, Wangi, Zhang, Hao, Zang, Cheng and Zhang	Predictive Mutation Testing	2016	ACM	International Symposium on Software Testing and Analysis (ISSTA)
[201]	Zhu, Panichella and Zaidman	Speeding-Up Mutation Testing via Data Compression and State Infection	2017	IEEE	International Workshop on Mutation Analysis (Mutation)
[202]	Zhu, Panichella and Zaidman	An Investigation of Compression Techniques to Speed up Mutation Testing	2018	IEEE	International Conference on Software Testing, Verification and Validation (ICST)

Table A.14: Subsumed studies (1/2) (Id = Reference ID).

Updated / Extended Studies				Subsumed Studies			
ID	Author	Year	Title	ID	Author	Year	Title
[107]	Krauser et al.	1991	High Performance Software Testing on SIMD Machines	[106]	Krauser et al.	1988	High Performance Testing on SIMD Machines
[133]	Offutt and Lee	1994	An Empirical Evaluation of Weak Mutation	[132]	Offutt and Lee	1991	How Strong is Weak Mutation?
[182]	Untch et al.	1993	Mutation Analysis Using Mutant Schemata	[179]	Untch	1992	Mutation-based Software Testing Using Program Schemata
[190]	Wong and Mathur	1995	Reducing the Cost of Mutation Testing: An Empirical Study	[122]	Mathur and Wong	1993	Evaluation of the Cost of Alternative Mutation Testing Strategies
[190]	Wong and Mathur	1995	Reducing the Cost of Mutation Testing: An Empirical Study	[191]	Wong et al.	1995	Mutation Versus All-uses: An Empirical Evaluation of Cost, Strength and Effectiveness
[136]	Offutt and Pan	1997	Automatically Detecting Equivalent Mutants and Infeasible Paths	[135]	Offutt and Pan	1996	Detecting Equivalent Mutants and the Feasible Path Problem
[8]	Alexander et al.	2002	Mutation of Java Objects	[17]	Bieman et al.	2001	A Technique for Mutation of Java Objects
[22]	Bogacki and Walter	2006a	Aspect-oriented Response Injection: an Alternative to Classical Mutation Testing	[23]	Bogacki and Walter	2006b	Evaluation of Test Code Quality with Aspect-Oriented Mutations
[49]	Domínguez-Jiménez et al.	2009a	GAmera: An Automatic Mutant Generation System for WS-BPEL Compositions	[51]	Domínguez-Jiménez et al.	2009b	A Framework for Mutant Genetic Generation for WS-BPEL
[150]	Papadakis and Malevris	2011b	Automatically Performing Weak Mutation with the Aid of Symbolic Execution, Concolic Testing and Search-based Testing	[152]	Papadakis et al.	2010	Towards Automating the Generation of Mutation Tests
[61]	Fraser and Zeller	2012	Mutation-Driven Generation of Oracles and Unit Tests	[60]	Fraser and Zeller	2010	Mutation-Driven Generation of Oracles and Unit Tests
[170]	Schuler and Zeller	2013	Covering and Uncovering Equivalent Mutants	[169]	Schuler and Zeller	2010	(Un-)Covering Equivalent Mutants
[94]	Kaminski et al.	2013	Improving logic-based testing	[93]	Kaminski et al.	2011a	Better Predicate Testing
[164]	Reales and Polo	2014	Reducing Mutation Costs Through Uncovered Mutants	[162]	Reales and Polo	2012	Mutant Execution Cost Reduction: Through MUSIC (Mutant Schema Improved with Extra Code)
[90]	Just and Schweiggert	2015	Higher Accuracy and Lower Run Time: Efficient Mutation Analysis Using Non-redundant Mutation Operators	[87]	Just et al.	2012a	Do Redundant Mutants Affect the Effectiveness and Efficiency of Mutation Analysis?
[6]	Aichernig et al.	2013	Incremental Refinement Checking for Test Case Generation	[5]	Aichernig and Jöbstl	2012	Efficient Refinement Checking for Model-Based Mutation Testing
[103]	Kintis et al.	2015	Employing Second-order Mutation for Isolating First-order Equivalent Mutants	[102]	Kintis et al.	2012	Isolating First Order Equivalent Mutants via Second Order Mutation
[84]	Just et al.	2014a	Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States	[83]	Just	2014	The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java
[100]	Kintis et al.	2017	Detecting Trivial Mutant Equivalences via Compiler Optimisations	[143]	Papadakis et al.	2015	Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique
[65]	Gopinath et al.	2017	Mutation Reduction Strategies Considered Harmful	[66]	Gopinath et al.	2016a	On the Limits of Mutation Reduction Strategies

Table A.15: Subsumed studies (2/2) (Id = Reference ID).

Updated / Extended Studies				Subsumed Studies			
ID	Author	Year	Title	ID	Author	Year	Title
[156]	Petrović and Ivanković	2018	State of Mutation Testing at Google	[157]	Petrović et al.	2018	An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions
[126]	McMinn et al.	2018	Automatic Detection and Removal of Ineffective Mutants for the Mutation Analysis of Relational Database Schemas	[194]	Wright et al.	2014	The Impact of Equivalent, Redundant and Quasi Mutants on Database Schema Mutation Analysis