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

Alessandro Viola Pizzoleto\textsuperscript{a}, Fabiano Cutigi Ferrari\textsuperscript{a,b,1,*}, Jeff Offutt\textsuperscript{b}, Leo Fernandes\textsuperscript{c}, Márcio Ribeiro\textsuperscript{d}

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

\section*{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.

\textbf{Keywords:} Mutation analysis, mutation testing, cost reduction, systematic review

\section{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

\textsuperscript{*}Corresponding author

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

\textsuperscript{1}Was a visiting researcher at George Mason University, Fairfax, VA, USA, in 2017 and 2018.

Preprint submitted to Journal of Systems and Software July 24, 2019
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.

\[
MS = \frac{K}{M - E}
\]

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 \land (I1 \lor I2)). We discarded studies that passed E1 or E2 (E1 \lor 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:

\[
\text{("mutation testing" or "mutation analysis" or "mutant analysis") and ("cost reduction" or "sufficient operator" or "sufficient mutation" or "constrained mutation" or "selective mutation" or "weak mutation" or "random selection" or "random mutation" or "random mutants" or "equivalent mutant" or "equivalent mutation")}
\]

The search string, sometimes customized for specific databases, was submitted to the search engines of the following databases: IEEE Xplore, ACM Digital Library, Elsevier ScienceDirect, Springer SpringerLink, and Wiley Online Library. 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

---

3The 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.


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

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

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

8http://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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - Automatic</td>
<td>467</td>
<td>27</td>
<td>612</td>
<td>19</td>
<td>606</td>
<td>12</td>
<td>494</td>
<td>13</td>
<td>771</td>
<td>13</td>
<td>19</td>
<td>13</td>
<td>2559</td>
<td>84</td>
</tr>
<tr>
<td>(Apr, 2016)</td>
<td>8.88%</td>
<td>17.65%</td>
<td>11.64%</td>
<td>12.42%</td>
<td>11.52%</td>
<td>7.84%</td>
<td>8.82%</td>
<td>0.00%</td>
<td>14.66%</td>
<td>8.50%</td>
<td>0.36%</td>
<td>8.50%</td>
<td>55.89%</td>
<td>54.90%</td>
</tr>
<tr>
<td>2 - Automatic</td>
<td>303</td>
<td>8</td>
<td>642</td>
<td>10</td>
<td>706</td>
<td>1</td>
<td>78</td>
<td>0</td>
<td>776</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2105</td>
<td>20</td>
</tr>
<tr>
<td>(Oct, 2017)</td>
<td>5.76%</td>
<td>5.23%</td>
<td>12.21%</td>
<td>6.54%</td>
<td>13.42%</td>
<td>0.65%</td>
<td>1.48%</td>
<td>0.00%</td>
<td>7.15%</td>
<td>0.65%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>40.03%</td>
<td>13.07%</td>
</tr>
<tr>
<td>3 - Snowballing</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>72</td>
<td>17</td>
</tr>
<tr>
<td>(Oct, 2017)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>72</td>
<td>17</td>
</tr>
<tr>
<td>4 - Author</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>135</td>
<td>25</td>
</tr>
<tr>
<td>Survey</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2.57%</td>
<td>16.34%</td>
</tr>
<tr>
<td>(Oct-Dec, 2017)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>135</td>
<td>25</td>
</tr>
<tr>
<td>5 - Update</td>
<td>8</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>8250</td>
<td>153</td>
</tr>
<tr>
<td>(Aug, 2018)</td>
<td>1.37%</td>
<td>11.11%</td>
<td>12.21%</td>
<td>6.54%</td>
<td>13.42%</td>
<td>0.65%</td>
<td>1.48%</td>
<td>0.00%</td>
<td>7.15%</td>
<td>0.65%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>40.03%</td>
<td>13.07%</td>
</tr>
<tr>
<td>TOTAL</td>
<td>770</td>
<td>35</td>
<td>1294</td>
<td>20</td>
<td>1312</td>
<td>13</td>
<td>542</td>
<td>0</td>
<td>1147</td>
<td>14</td>
<td>19</td>
<td>13</td>
<td>8250</td>
<td>153</td>
</tr>
<tr>
<td></td>
<td>14.64%</td>
<td>22.88%</td>
<td>23.84%</td>
<td>18.95%</td>
<td>24.05%</td>
<td>8.50%</td>
<td>10.31%</td>
<td>0.00%</td>
<td>21.81%</td>
<td>9.15%</td>
<td>0.36%</td>
<td>8.50%</td>
<td>0.15%</td>
<td>4.58%</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Database</th>
<th># of Studies</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE</td>
<td>58</td>
<td>37.91%</td>
</tr>
<tr>
<td>ACM</td>
<td>33</td>
<td>21.57%</td>
</tr>
<tr>
<td>Springer</td>
<td>15</td>
<td>9.80%</td>
</tr>
<tr>
<td>Elsevier</td>
<td>17</td>
<td>11.11%</td>
</tr>
<tr>
<td>Wiley</td>
<td>20</td>
<td>13.07%</td>
</tr>
<tr>
<td>Others</td>
<td>10</td>
<td>6.54%</td>
</tr>
<tr>
<td>Total</td>
<td>153</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

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.

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 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
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: (▽) 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: (≡) 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: (▷) 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: (⊘) 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: \((\tau)\) 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.

![Figure 4: Relationships among primary goals.](image)

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.

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.
**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.

<table>
<thead>
<tr>
<th>Technique</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-4 - Firm Mutation</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-18 - Execution Trace Analysis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-7 - Minimization and Prioritization</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-21 - Minimal Mutation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-9 - Constrained Mutation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-19 - Model-based Mutation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-13 - Serial Execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-20 - State-based analysis</td>
<td></td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-8 - Compiler Optimization</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-17 - One-op Mutation</td>
<td></td>
<td></td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-5 - Parallel execution</td>
<td></td>
<td></td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-14 - Sufficient Operators</td>
<td></td>
<td></td>
<td>7</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-1 - Random Mutation</td>
<td></td>
<td></td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-3 - Weak Mutation</td>
<td></td>
<td></td>
<td>3</td>
<td>7</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-10 - Metamutants</td>
<td></td>
<td></td>
<td>6</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-2 - Higher Order Mutation</td>
<td></td>
<td>10</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-6 - Data-flow analysis</td>
<td></td>
<td>13</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-12 - Selective Mutation</td>
<td></td>
<td>12</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-16 - Evolutionary algorithms</td>
<td></td>
<td>17</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-11 - Control-flow analysis</td>
<td></td>
<td>22</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-15 - Optimization of generation,</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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 \((PG-I)\), Gopinath et al. \cite{65} applied
Table 3: List of studies that applied combined techniques per year (years with no studies are omitted).

<table>
<thead>
<tr>
<th>Year</th>
<th>Studies and Respectiv e Cost Reduction Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>1990</td>
<td>[119] (T-6, T-3)</td>
</tr>
<tr>
<td>1993</td>
<td>[187] (T-13, T-10)</td>
</tr>
<tr>
<td>1994</td>
<td>[129] (T-6, T-8)</td>
</tr>
<tr>
<td>1995</td>
<td>[100] (T-1, T-9)</td>
</tr>
<tr>
<td>2000</td>
<td>[70] (T-11, T-6); [80] (T-1, T-5)</td>
</tr>
<tr>
<td>2006</td>
<td>[34] (T-6, T-11)</td>
</tr>
<tr>
<td>2007</td>
<td>[178] (T-7, T-12)</td>
</tr>
<tr>
<td>2009</td>
<td>[168] (T-6, T-11); [81] (T-11, T-16)</td>
</tr>
<tr>
<td>2010</td>
<td>[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)</td>
</tr>
<tr>
<td>2011</td>
<td>[86] (T-15, T-8); [92] (T-11, T-2); [150] (T-8, T-11, T-10, T-16)</td>
</tr>
<tr>
<td>2012</td>
<td>[135] (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)</td>
</tr>
<tr>
<td>2013</td>
<td>[165] (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)</td>
</tr>
<tr>
<td>2014</td>
<td>[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)</td>
</tr>
<tr>
<td>2015</td>
<td>[165] (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)</td>
</tr>
<tr>
<td>2016</td>
<td>[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)</td>
</tr>
<tr>
<td>2017</td>
<td>[145] (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)</td>
</tr>
<tr>
<td>2018</td>
<td>[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)</td>
</tr>
</tbody>
</table>

three techniques: \textit{t-1} (random mutation), \textit{T-9} (constrained mutation) and \textit{T-12} (selective mutation). Therefore, Gopinath et al.’s study is listed three times for \textit{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 \textit{T-6} (Data-flow analysis) and \textit{T-11} (Control-flow analysis) to reduce the number of test cases or the number of executions (\textit{PG-4}), and to avoid the creation of certain mutants (\textit{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 (\textit{e.g.} Just et al.’s [88] study is listed two times for \textit{T-6} and two times for \textit{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]\footnote{http://goo.gl/edyF1n–last accessed November 2018.}. 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
Table 4: Mapping between primary goals and cost reduction techniques.

<table>
<thead>
<tr>
<th>Primary Goal</th>
<th>Cost Reduction Techniques and Respective Studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>PG-1</td>
<td>Reducing the number of mutants</td>
</tr>
<tr>
<td>T-1</td>
<td>[45, 65, 108, 147, 153, 156, 190, 196, 197]</td>
</tr>
<tr>
<td>T-2</td>
<td>[42, 71, 96, 101, 112, 117, 141, 147, 158, 165, 166]</td>
</tr>
<tr>
<td>T-3</td>
<td>[101, 201, 202]</td>
</tr>
<tr>
<td>T-4</td>
<td>[43, 79, 91, 166, 186, 201, 202]</td>
</tr>
<tr>
<td>T-5</td>
<td>[1, 3, 31, 33, 30, 71, 81, 128, 140, 161, 174]</td>
</tr>
<tr>
<td>T-6</td>
<td>[7, 155]</td>
</tr>
<tr>
<td>T-7</td>
<td>[178]</td>
</tr>
<tr>
<td>T-8</td>
<td>[31]</td>
</tr>
<tr>
<td>T-9</td>
<td>[65, 156, 189, 197]</td>
</tr>
<tr>
<td>T-10</td>
<td>[13, 29, 109, 171, 172, 183, 197]</td>
</tr>
<tr>
<td>T-11</td>
<td>[25, 43, 79, 91, 166, 186, 201, 202]</td>
</tr>
<tr>
<td>T-12</td>
<td>[1, 3, 31, 33, 30, 71, 81, 128, 140, 161, 174]</td>
</tr>
<tr>
<td>T-13</td>
<td>[31]</td>
</tr>
<tr>
<td>T-14</td>
<td>[6, 30, 39, 42, 108, 190]</td>
</tr>
<tr>
<td>T-15</td>
<td>[19]</td>
</tr>
<tr>
<td>T-16</td>
<td>[4, 6]</td>
</tr>
<tr>
<td>T-17</td>
<td>[9, 89, 108]</td>
</tr>
<tr>
<td>T-18</td>
<td>[46, 68, 71, 92]</td>
</tr>
<tr>
<td>T-19</td>
<td>[59, 114, 201, 202]</td>
</tr>
<tr>
<td>T-20</td>
<td>[62, 63, 84, 185]</td>
</tr>
<tr>
<td>T-21</td>
<td>[6]</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Primary Goal</th>
<th>Cost Reduction Techniques and Respective Studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-1</td>
<td>Random Mutation</td>
</tr>
<tr>
<td>PG-1</td>
<td>[45, 65, 108, 147, 153, 156, 190, 196, 197]</td>
</tr>
<tr>
<td>T-2</td>
<td>Higher Order Mutation</td>
</tr>
<tr>
<td>PG-1</td>
<td>[1, 42, 71, 96, 101, 112, 117, 141, 147, 158, 165, 166]</td>
</tr>
<tr>
<td>PG-2</td>
<td>[103]</td>
</tr>
<tr>
<td>PG-3</td>
<td>[46]</td>
</tr>
<tr>
<td>PG-4</td>
<td>[46, 68, 71, 92]</td>
</tr>
<tr>
<td>T-3</td>
<td>Weak Mutation</td>
</tr>
<tr>
<td>PG-1</td>
<td>[101, 201, 202]</td>
</tr>
<tr>
<td>PG-2</td>
<td>[41, 133]</td>
</tr>
<tr>
<td>PG-3</td>
<td>[47, 96, 133, 150]</td>
</tr>
<tr>
<td>PG-4</td>
<td>[59, 114, 201, 202]</td>
</tr>
<tr>
<td>PG-5</td>
<td>[119]</td>
</tr>
<tr>
<td>PG-6</td>
<td>[160, 200]</td>
</tr>
<tr>
<td>T-4</td>
<td>Firm Mutation</td>
</tr>
<tr>
<td>PG-3</td>
<td>[80]</td>
</tr>
<tr>
<td>T-5</td>
<td>Parallel execution</td>
</tr>
<tr>
<td>PG-3</td>
<td>[27, 80, 101, 111, 137, 163, 185]</td>
</tr>
<tr>
<td>T-6</td>
<td>Data-flow analysis</td>
</tr>
<tr>
<td>PG-1</td>
<td>[178]</td>
</tr>
<tr>
<td>PG-2</td>
<td>[167, 178]</td>
</tr>
<tr>
<td>PG-3</td>
<td>[114]</td>
</tr>
<tr>
<td>PG-4</td>
<td>[59, 114, 201, 202]</td>
</tr>
<tr>
<td>PG-5</td>
<td>[32, 88, 98, 119]</td>
</tr>
<tr>
<td>PG-6</td>
<td>[61]</td>
</tr>
<tr>
<td>T-7</td>
<td>Minimization and Prioritization of test sets</td>
</tr>
<tr>
<td>PG-3</td>
<td>[47, 96, 133, 150]</td>
</tr>
<tr>
<td>T-8</td>
<td>Compiler Optimization</td>
</tr>
<tr>
<td>PG-1</td>
<td>[65, 108, 147, 153, 156, 190, 196, 197]</td>
</tr>
<tr>
<td>T-9</td>
<td>[103]</td>
</tr>
<tr>
<td>PG-3</td>
<td>[46]</td>
</tr>
<tr>
<td>PG-4</td>
<td>[46, 68, 71, 92]</td>
</tr>
<tr>
<td>T-10</td>
<td>Metamutants</td>
</tr>
<tr>
<td>PG-3</td>
<td>[103]</td>
</tr>
<tr>
<td>PG-4</td>
<td>[167, 178]</td>
</tr>
<tr>
<td>T-11</td>
<td>Constrained Mutation</td>
</tr>
<tr>
<td>PG-1</td>
<td>[103]</td>
</tr>
<tr>
<td>PG-2</td>
<td>[31]</td>
</tr>
<tr>
<td>PG-3</td>
<td>[46, 133, 150]</td>
</tr>
<tr>
<td>PG-4</td>
<td>[59, 114, 201, 202]</td>
</tr>
<tr>
<td>PG-5</td>
<td>[59, 114, 201, 202]</td>
</tr>
<tr>
<td>PG-6</td>
<td>[160, 200]</td>
</tr>
<tr>
<td>T-12</td>
<td>Selective Mutation</td>
</tr>
<tr>
<td>PG-1</td>
<td>[167, 178]</td>
</tr>
<tr>
<td>T-13</td>
<td>Serial Execution</td>
</tr>
<tr>
<td>PG-3</td>
<td>[14, 61, 72, 124, 149, 190]</td>
</tr>
<tr>
<td>T-14</td>
<td>State-based analysis</td>
</tr>
<tr>
<td>PG-3</td>
<td>[52, 88, 88, 119]</td>
</tr>
<tr>
<td>T-15</td>
<td>Optimization of generation, execution and analysis of mutants</td>
</tr>
<tr>
<td>PG-1</td>
<td>[79, 91, 150, 156, 190, 196, 197]</td>
</tr>
<tr>
<td>T-16</td>
<td>Evolutionary algorithms</td>
</tr>
<tr>
<td>PG-1</td>
<td>[79, 91, 150, 156, 190, 196, 197]</td>
</tr>
<tr>
<td>T-17</td>
<td>One-tap Mutation</td>
</tr>
<tr>
<td>PG-1</td>
<td>[28, 30, 39, 42, 108, 190]</td>
</tr>
<tr>
<td>T-18</td>
<td>Execution Trace Analysis</td>
</tr>
<tr>
<td>PG-1</td>
<td>[28, 30, 39, 42, 108, 190]</td>
</tr>
<tr>
<td>T-19</td>
<td>Model-based mutation</td>
</tr>
<tr>
<td>PG-2</td>
<td>[46]</td>
</tr>
<tr>
<td>PG-3</td>
<td>[62, 63, 84, 185]</td>
</tr>
<tr>
<td>PG-4</td>
<td>[4]</td>
</tr>
<tr>
<td>T-20</td>
<td>State-based analysis</td>
</tr>
<tr>
<td>PG-3</td>
<td>[62, 63, 84, 185]</td>
</tr>
<tr>
<td>PG-4</td>
<td>[4]</td>
</tr>
<tr>
<td>T-21</td>
<td>Minimal Mutation</td>
</tr>
<tr>
<td>PG-1</td>
<td>[9, 89, 108]</td>
</tr>
</tbody>
</table>

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
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.


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, MTau, 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 Halas [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 com-
puter. 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 Fleystgakker [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 refactoring 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.

<table>
<thead>
<tr>
<th>ID</th>
<th>Metric name</th>
<th>Intent</th>
<th>Studies that collected the metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-1</td>
<td>Number of test cases required to achieve mutant coverage</td>
<td>Estimate the number of test cases that need to be created (automatically or manually) or executed.</td>
<td>[20, 25, 26, 28, 30, 34, 41, 44, 45, 71, 81, 92, 112, 113, 140, 147, 151, 165, 167, 178, 189, 190, 191, 201]</td>
</tr>
<tr>
<td>M-2</td>
<td>Number of mutants to be executed</td>
<td>Estimate the number of mutants that need to be executed.</td>
<td>[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, 99, 90, 91, 94, 95, 109, 112, 117, 118, 119, 129, 131, 136, 140, 141, 147, 153, 155, 160, 161, 165, 166, 171, 172, 174, 175, 176, 178, 180, 183, 186, 190, 191, 196, 197, 201, 202]</td>
</tr>
<tr>
<td>M-3</td>
<td>Mutant execution speedup</td>
<td>Estimate how much time is required for mutant execution.</td>
<td>[22, 26, 27, 32, 43, 45, 46, 52, 57, 62, 63, 67, 68, 90, 96, 107, 111, 114, 115, 117, 126, 133, 137, 150, 163, 164, 161, 162, 185, 187, 193, 195, 196, 201, 202]</td>
</tr>
<tr>
<td>M-4</td>
<td>Mutant execution efficiency</td>
<td>Estimate the performance gains or losses when two or more execution configurations (e.g. processor and network infrastructure) are compared.</td>
<td>[137]</td>
</tr>
<tr>
<td>M-5</td>
<td>Test generation speedup</td>
<td>Estimate how much time is required for test generation.</td>
<td>[98, 113]</td>
</tr>
<tr>
<td>M-6</td>
<td>Number of equivalent mutants automatically detected</td>
<td>Estimate the number of mutants that need to be handled during mutant analysis.</td>
<td>[10, 47, 53, 56, 97, 99, 100, 118, 126, 129, 134, 136, 142, 145, 154, 168, 194]</td>
</tr>
<tr>
<td>M-7</td>
<td>Number of cycles (or generations) in evolutionary algorithms</td>
<td>Estimate the effort required to run a specific evolutionary algorithm.</td>
<td>[14, 15]</td>
</tr>
<tr>
<td>M-8</td>
<td>Mutant compilation speedup</td>
<td>Estimate how much time is required for mutant compilation.</td>
<td>[82, 83, 86, 115]</td>
</tr>
<tr>
<td>M-9</td>
<td>Mutant generation speedup</td>
<td>Estimate how much time is required for mutant generation.</td>
<td>[80, 86, 96, 115]</td>
</tr>
<tr>
<td>M-10</td>
<td>Completeness of generated test suite</td>
<td>Estimate the effort required to evolve the current set suite to achieve the intended mutant coverage.</td>
<td>[91, 124, 146, 151, 200]</td>
</tr>
<tr>
<td>M-11</td>
<td>Number of killable mutants automatically detected</td>
<td>Estimate the number of mutants that need to be handled during mutant analysis.</td>
<td>[47, 75, 154, 168]</td>
</tr>
<tr>
<td>M-12</td>
<td>Number of equivalent mutants generated</td>
<td>Estimate the number of mutants that need to be handled during mutant analysis.</td>
<td>[28, 29, 30, 39, 77, 95, 98, 101, 117, 147, 158, 190]</td>
</tr>
<tr>
<td>M-13</td>
<td>Number of cycles in test generation algorithms</td>
<td>Estimate the effort required to run a specific test generation algorithm.</td>
<td>[148]</td>
</tr>
<tr>
<td>M-14</td>
<td>Mutant analysis speedup</td>
<td>Estimate how much time is required for mutant analysis.</td>
<td>[6, 47, 117, 175, 194]</td>
</tr>
<tr>
<td>M-15</td>
<td>Probability to kill mutants</td>
<td>To obtain a probability for mutant coverage based on the available test set.</td>
<td>[155]</td>
</tr>
<tr>
<td>M-16</td>
<td>Number of test case executions</td>
<td>Estimate the number of test case executions for a given mutant set.</td>
<td>[14, 26, 43, 64, 90, 114, 164, 167, 195, 199]</td>
</tr>
<tr>
<td>M-17</td>
<td>Number of duplicated mutants automatically detected</td>
<td>Estimate the number of mutants that need to be handled during mutant analysis.</td>
<td>[63, 100, 126]</td>
</tr>
<tr>
<td>M-18</td>
<td>Variation in mutation score</td>
<td>Estimate gains and losses with respect to mutation score.</td>
<td>[177]</td>
</tr>
</tbody>
</table>
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. Sahinoglu 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.

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

---

10Aichernig 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 specifically 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 Pizzeleto 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**


Informatica 8, 31–45.


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


[98] Kintis, M., Malevris, N., 2015. MEDIC: A Static Analysis Framework for Equivalent Mutant Identi-


53


55
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)

<table>
<thead>
<tr>
<th>Id</th>
<th>Author</th>
<th>Title</th>
<th>Year</th>
<th>Publisher</th>
<th>Journal/Event</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>and Computer</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Science Press</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>International</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Conference on</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Quality Software</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(QSIC)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Schlick and Schmidt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ch and Saake</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rones</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>cenai</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>on Software</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Engineering</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>on Software</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Engineering</td>
<td></td>
</tr>
<tr>
<td>[22]</td>
<td>Delamaro, Deng, Durelli, Li and Offutt</td>
<td>Experimental Evaluation of SDL and One-Op Mutation for C</td>
<td>2014a</td>
<td>IEEE</td>
<td>International Conference on Software Testing, Verification and Validation (ICST)</td>
</tr>
<tr>
<td>[23]</td>
<td>Delamaro, Offutt and Ammann</td>
<td>Designing Deletion Mutation Operators</td>
<td>2014b</td>
<td>IEEE</td>
<td>International Conference on Software Testing, Verification and Validation (ICST)</td>
</tr>
</tbody>
</table>

57
<table>
<thead>
<tr>
<th>Id</th>
<th>Author</th>
<th>Title</th>
<th>Year</th>
<th>Publisher</th>
<th>Journal/Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>Delamaro, Deng, Li, Utreli and Offutt</td>
<td>Growing a Reduced Set of Mutation Operators</td>
<td>2014b</td>
<td>IEEE</td>
<td>Brazilian Symposium on Software Engineering (SBES)</td>
</tr>
<tr>
<td>33</td>
<td>Delgado-Pérez, Medina-Bullo, Segura, García-Domínguez and de la Oya</td>
<td>GiGAn: Evolutionary Mutation Testing for C++ Object-Oriented Systems</td>
<td>2017b</td>
<td>ACM</td>
<td>ACM Symposium on Applied Computing (SAC)</td>
</tr>
<tr>
<td>34</td>
<td>Delgado-Pérez, Segura and Medina-Bullo</td>
<td>Assessment of C++ Object-Oriented Mutation Operators: A Selective Mutation Approach</td>
<td>2017c</td>
<td>Wiley</td>
<td>Software Testing, Verification and Reliability</td>
</tr>
<tr>
<td>32</td>
<td>Delgado-Pérez, Medina-Bullo, Palomo-Lopez, García-Domínguez and Domínguez-Jiménez</td>
<td>Assessment of Class Mutation Operators for C++ with the MuCPP Mutation System</td>
<td>2017a</td>
<td>Elsevier</td>
<td>Information and Software Technology</td>
</tr>
<tr>
<td>31</td>
<td>Delgado-Pérez, Medina-Bullo and Meraño</td>
<td>Using Evolutionary Computation to Improve Mutation Testing</td>
<td>2017</td>
<td>Springer</td>
<td>International Work-Conference on Artificial Neural Networks (IWANN)</td>
</tr>
<tr>
<td>38</td>
<td>Delmillo and Offutt</td>
<td>Experimental Results from an Automatic Test Case Generator</td>
<td>1993</td>
<td>ACM</td>
<td>ACM Transactions on Software Engineering and Methodology</td>
</tr>
<tr>
<td>41</td>
<td>Derezińska</td>
<td>A Quality Estimation of Mutation Clustering in C# Programs</td>
<td>2013</td>
<td>Springer</td>
<td>International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)</td>
</tr>
<tr>
<td>43</td>
<td>Derezińska and Halas</td>
<td>Improving Mutation Testing Process of Python Programs</td>
<td>2015</td>
<td>Springer</td>
<td>Computer Science Online Conference Software Engineering in Intelligent Systems (COSO)</td>
</tr>
<tr>
<td>44</td>
<td>Derezińska</td>
<td>Evaluation of Deletion Mutation Operators in Mutation Testing of C# Programs</td>
<td>2016</td>
<td>Springer</td>
<td>International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)</td>
</tr>
<tr>
<td>45</td>
<td>Derezińska and Rudnik</td>
<td>Evaluation of Mutant Sampling Criteria in Object-Oriented Mutation Testing</td>
<td>2017</td>
<td>Polskie Towarzystwo Informatyczne</td>
<td>Federated Conference on Computer Science and Information Systems (ACSIS)</td>
</tr>
<tr>
<td>46</td>
<td>Devroey, Perrouin, Papadakis, Legay Schobbens and Heymans</td>
<td>Featured Model-based Mutation Analysis</td>
<td>2016</td>
<td>ACM</td>
<td>International Conference on Software Engineering (ICSE)</td>
</tr>
<tr>
<td>50</td>
<td>Domínguez-Jiménez, Estero-Botaro, García-Domínguez and Medina-Bullo</td>
<td>Evolutionary Mutation Testing</td>
<td>2011</td>
<td>Elsevier</td>
<td>Information and Software Technology</td>
</tr>
<tr>
<td>54</td>
<td>Utreli, Offutt and Delamaro</td>
<td>Toward Harnessing High-Level Language Virtual Machines for Further Speeding Up Weak Mutation Testing</td>
<td>2012</td>
<td>IEEE</td>
<td>International Conference on Software Testing, Verification and Validation (ICST)</td>
</tr>
</tbody>
</table>
Table A.9: Selected Studies (3/7) (Id = Reference ID)

<table>
<thead>
<tr>
<th>Id</th>
<th>Author</th>
<th>Title</th>
<th>Year</th>
<th>Publisher</th>
<th>Journal/Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>[53]</td>
<td>Fernandes, Ribeiro, Carvalho, Gheyi, Mongiow, Santos, Cavalcanti, Ferrari and Maldonado</td>
<td>Avoiding Useless Mutants</td>
<td>2017</td>
<td>ACM</td>
<td>International Conference on Generative Programming: Concepts and Experiences (GPCE)</td>
</tr>
<tr>
<td>[56]</td>
<td>Ferrari, Rashid and Maldonado</td>
<td>Towards the Practical Mutation Testing of AspectJ Programs</td>
<td>2013</td>
<td>Elsevier</td>
<td>Science of Computer Programming</td>
</tr>
<tr>
<td>[66]</td>
<td>Gopinath, Mathis and Zeller</td>
<td>If You Can’t Kill a Supermutant, You Have a Problem</td>
<td>2018</td>
<td>IEEE</td>
<td>International Workshop on Mutation Analysis (Mutation)</td>
</tr>
<tr>
<td>[74]</td>
<td>Hierons, Harman and Danke</td>
<td>Using Program Slicing to Assist in the Detection of Equivalent Mutants</td>
<td>1999</td>
<td>Wiley</td>
<td>Software Testing, Verification and Reliability</td>
</tr>
<tr>
<td>[75]</td>
<td>Henant, Papadakis and Le Traon</td>
<td>Mutation-based generation of Software Product Line Test Configurations</td>
<td>2014</td>
<td>Springer</td>
<td>International Symposium on Search-Based Software Engineering (SSBSE)</td>
</tr>
<tr>
<td>[74]</td>
<td>Holing, Banescu, Popescu, Petroska and Pietraniker</td>
<td>Nequack: Assessing Mutation Score Confidence</td>
<td>2016</td>
<td>IEEE</td>
<td>International Workshop on Mutation Analysis (Mutation)</td>
</tr>
<tr>
<td>[77]</td>
<td>Ho, Liand Offutt</td>
<td>An Analysis of OO Mutation Operators</td>
<td>2011</td>
<td>IEEE</td>
<td>International Workshop on Mutation Analysis (Mutation)</td>
</tr>
<tr>
<td>[78]</td>
<td>Iida and Takada</td>
<td>Reducing Mutants with Mutant Killable Precondition</td>
<td>2017</td>
<td>IEEE</td>
<td>International Workshop on Mutation Analysis (Mutation)</td>
</tr>
<tr>
<td>[79]</td>
<td>Inozemtseva, Hemmati and Holmes</td>
<td>Using Fault History to Improve Mutation Reduction</td>
<td>2013</td>
<td>ACM</td>
<td>International Symposium on the Foundations of Software Engineering (FSE)</td>
</tr>
<tr>
<td>[81]</td>
<td>Ji, Chen, Xu and Zhao</td>
<td>A Novel Method of Mutation Clustering Based on Domain Analysis</td>
<td>2009</td>
<td>Knowledge Systems Institute Graduate School</td>
<td>International Conference on Software Engineering and Knowledge Engineering (SEKE)</td>
</tr>
<tr>
<td>Id</td>
<td>Author</td>
<td>Title</td>
<td>Year</td>
<td>Publisher</td>
<td>Journal/Event</td>
</tr>
<tr>
<td>------</td>
<td>-------------------------------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------</td>
</tr>
<tr>
<td>88</td>
<td>Kurtz, Kaphammer and Schwartz</td>
<td>Using Non-Redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis</td>
<td>2012</td>
<td>IEEE</td>
<td>International Symposium on Software Reliability Engineering (ISSRE)</td>
</tr>
<tr>
<td>84</td>
<td>Just, Ernst and Fraser</td>
<td>Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States</td>
<td>2011</td>
<td>ACM</td>
<td>International Symposium on Software Testing and Analysis (SSST)</td>
</tr>
<tr>
<td>83</td>
<td>Just and Schwiegert</td>
<td>Higher Accuracy and Lower Non-Time: Efficient Mutation Analysis Using Non-redundant Mutation Operators</td>
<td>2015</td>
<td>Wiley</td>
<td>Software Testing, Verification and Reliability</td>
</tr>
<tr>
<td>88</td>
<td>Just, Kurtz and Ammann</td>
<td>Inferring Mutant Utility from Program Context</td>
<td>2017</td>
<td>ACM</td>
<td>International Symposium on Software Testing and Analysis (SSST)</td>
</tr>
<tr>
<td>91</td>
<td>Kaminski and Ammann</td>
<td>Using a Fault Hierarchy to Improve the Efficiency of DNF Logic Mutation Testing</td>
<td>2009</td>
<td>IEEE</td>
<td>International Conference on Software Testing, Verification and Validation (ICST)</td>
</tr>
<tr>
<td>92</td>
<td>Kaminski and Ammann</td>
<td>Reducing Logic Test Set Size while Preserving Fault Detection</td>
<td>2011</td>
<td>Wiley</td>
<td>Software Testing, Verification and Reliability</td>
</tr>
<tr>
<td>93</td>
<td>Kaminski, Praphamontripong, Ammann and Offutt</td>
<td>A Logic Mutation Approach to Selective Mutation for Programs and Queries</td>
<td>2011b</td>
<td>Elsevier</td>
<td>Information and Software Technology</td>
</tr>
<tr>
<td>94</td>
<td>Kaminski, Ammann and Offutt</td>
<td>Improving logic-based testing</td>
<td>2013</td>
<td>Elsevier</td>
<td>Journal of Systems and Software</td>
</tr>
<tr>
<td>96</td>
<td>Kim, Ma and Kwon</td>
<td>Combining Weak and Strong Mutation for a Non-interpretable Java Mutation System</td>
<td>2012</td>
<td>Wiley</td>
<td>Software Testing, Verification and Reliability</td>
</tr>
<tr>
<td>101</td>
<td>Kintis, Papadakis and Malevris</td>
<td>Evaluating Mutation Testing Alternatives: A Collateral Experiment</td>
<td>2010</td>
<td>IEEE</td>
<td>Asia-Pacific Software Engineering Conference (APSEC)</td>
</tr>
<tr>
<td>102</td>
<td>Kintis and Malevris</td>
<td>Identifying More Equivalent Mutants via Code Similarity</td>
<td>2013</td>
<td>IEEE</td>
<td>Asia-Pacific Software Engineering Conference (APSEC)</td>
</tr>
<tr>
<td>103</td>
<td>Kintis and Malevris</td>
<td>Using Data Flow Patterns for Equivalent Mutant Detection</td>
<td>2014</td>
<td>IEEE</td>
<td>International Workshop on Mutation Analysis (Mutation)</td>
</tr>
<tr>
<td>104</td>
<td>Kintis and Malevris</td>
<td>Employing Second-Order Mutation for Isolating First-order Equivalent Mutants</td>
<td>2015</td>
<td>Wiley</td>
<td>Software Testing, Verification and Reliability</td>
</tr>
<tr>
<td>105</td>
<td>Kintis and Malevris</td>
<td>MEDIC: A Static Analysis Framework for Equivalent Mutant Identification</td>
<td>2015</td>
<td>Elsevier</td>
<td>Information and Software Technology</td>
</tr>
<tr>
<td>106</td>
<td>Kintis, Papadakis, Ja, Malevris, Le Taon and Harman</td>
<td>Detecting Invalid Mutation Equivalences via Compiler Optimisations</td>
<td>2017</td>
<td>IEEE</td>
<td>Transactions on Software Engineering</td>
</tr>
<tr>
<td>109</td>
<td>Lacenda and Ferrari</td>
<td>Towards the Establishment of a Sufficient Set of Mutation Operators for AspectJ Programs</td>
<td>2014</td>
<td>SBC</td>
<td>Brazilian Symposium on Systematic and Automated Software Testing (SAST)</td>
</tr>
<tr>
<td>111</td>
<td>Li, West, Escalona and Durell</td>
<td>Mutation Testing in Practice Using Ruby</td>
<td>2015</td>
<td>IEEE</td>
<td>International Conference on Mutation Analysis (Mutation)</td>
</tr>
<tr>
<td>112</td>
<td>Lima, Guizzo, Vérgilio, Silva, Filho and Ehrenfried</td>
<td>Evaluating Different Strategies for Reduction of Mutation Testing Costs</td>
<td>2016</td>
<td>ACM</td>
<td>Brazilian Symposium on Systematic and Automated Software Testing (SAST)</td>
</tr>
<tr>
<td>113</td>
<td>Liu, Gao, Shan, Liu, Zhang and Sun</td>
<td>An Approach to Test Data Generation for Killing Multiple Mutants</td>
<td>2006</td>
<td>IEEE</td>
<td>International Conference on Software Maintenance (ICSM)</td>
</tr>
<tr>
<td>115</td>
<td>Ma, Offutt and Kwon</td>
<td>MuJava: An Automated Class Mutation System</td>
<td>2005</td>
<td>Wiley</td>
<td>Software Testing, Verification and Reliability</td>
</tr>
<tr>
<td>114</td>
<td>Ma and Kim</td>
<td>Mutation Testing Cost Reduction by Clustering Overlapped Mutants</td>
<td>2016</td>
<td>Elsevier</td>
<td>Journal of Systems and Software</td>
</tr>
<tr>
<td>117</td>
<td>Madeyski, Orzeszyna, Toltar and Józefi</td>
<td>Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation</td>
<td>2014</td>
<td>IEEE</td>
<td>Transactions on Software Engineering</td>
</tr>
<tr>
<td>Id</td>
<td>Author</td>
<td>Title</td>
<td>Year</td>
<td>Publisher</td>
<td>Journal/Event</td>
</tr>
<tr>
<td>------</td>
<td>------------------------------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>[158]</td>
<td>Marcuzzi, Baldin, Kosematov, Papadakis, Prevosto and Corresino</td>
<td>Time to Clean Your Test Objectives</td>
<td>2018</td>
<td>IEEE</td>
<td>International Conference on Software Engineering (ICSE)</td>
</tr>
<tr>
<td>[159]</td>
<td>Marshall, Hedley, Riddell and Hessell</td>
<td>Static Dataflow-aided Weak Mutation Analysis (SDAWM)</td>
<td>1990</td>
<td>Elsevier</td>
<td>Information and Software Technology</td>
</tr>
<tr>
<td>[160]</td>
<td>Reales and Polo</td>
<td>Parallel Mutation testing</td>
<td>2013</td>
<td>wiley</td>
<td>Software Testing, Verification and Reliability</td>
</tr>
<tr>
<td>[163]</td>
<td>McMinn, Kapthammer and Wright</td>
<td>Virtual Mutation Analysis on Relational Database Schemas</td>
<td>2015</td>
<td>IEEE</td>
<td>International Workshop on Automation of Software Test (AST)</td>
</tr>
<tr>
<td>[170]</td>
<td>Offutt, Rothermel and Zapf</td>
<td>An Experimental Evaluation of Selective Mutation</td>
<td>1993</td>
<td>IEEE</td>
<td>International Conference on Software Engineering (ICSE)</td>
</tr>
<tr>
<td>[174]</td>
<td>Offutt and Pan</td>
<td>Automatically Detecting Equivalent Mutants and Infeasible Paths</td>
<td>1997</td>
<td>ACM</td>
<td>ACM Transactions on Software Engineering and Methodology</td>
</tr>
<tr>
<td>[175]</td>
<td>Offutt, Jin and Pan</td>
<td>The Dynamic UDOM Reduction Procedure for Test Data Generation</td>
<td>1999</td>
<td>Wiley</td>
<td>Software Practice and Experience</td>
</tr>
<tr>
<td>[176]</td>
<td>Offutt, Ma and Kwon</td>
<td>The Class-Level Mutants of MuJava</td>
<td>2006</td>
<td>ACM</td>
<td>International Workshop on Automation of Software Test (AST)</td>
</tr>
<tr>
<td>[177]</td>
<td>Oliveira, Camilo-Junior and Veillet</td>
<td>A Convolutional Algorithm to Automatic Test Case Selection and Mutant in Mutation Testing</td>
<td>2013</td>
<td>IEEE</td>
<td>IEEE Congress on Evolutionary Computation (CEC)</td>
</tr>
<tr>
<td>[181]</td>
<td>Papadakis and Makræs</td>
<td>Automatic Mutation Test Case Generation via Dynamic Symbolic Execution</td>
<td>2010</td>
<td>IEEE</td>
<td>International Symposium on Software Reliability Engineering (ISSRE)</td>
</tr>
<tr>
<td>[182]</td>
<td>Papadakis and Makræs</td>
<td>Automatic Mutation based Test Data Generation</td>
<td>2011</td>
<td>ACM</td>
<td>Genetic and Evolutionary Computation Conference (GECCO)</td>
</tr>
</tbody>
</table>
Table A.12: Selected Studies (6/7) (Id = Reference ID)

<table>
<thead>
<tr>
<th>Id</th>
<th>Author</th>
<th>Title</th>
<th>Year</th>
<th>Publisher</th>
<th>Journal/Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>[151]</td>
<td>Papadakis and Malevris</td>
<td>Mutation Based Test Case Generation Via a Path Selection Strategy</td>
<td>2012</td>
<td>Elsevier</td>
<td>Information and Software Technology</td>
</tr>
<tr>
<td>[149]</td>
<td>Papadakis and Le Traon</td>
<td>Mutation Testing Strategies using Mutant Classification</td>
<td>2013</td>
<td>ACM</td>
<td>ACM Symposium on Applied Computing (SAC)</td>
</tr>
<tr>
<td>[142]</td>
<td>Papadakis, Delamaro and Le Traon</td>
<td>Mitigating the Effects of Equivalent Mutants with Mutant Classification Strategies</td>
<td>2014</td>
<td>Elsevier</td>
<td>Science of Computer Programming</td>
</tr>
<tr>
<td>[153]</td>
<td>Parsi, Murgia and Demeyer</td>
<td>Evaluating Random Mutant Selection at Class-level in Projects with Non-adequate Test Suites</td>
<td>2016</td>
<td>ACM</td>
<td>International on Evaluation and Assessment in Software Engineering (EASE)</td>
</tr>
<tr>
<td>[157]</td>
<td>Papamontopong and Offutt</td>
<td>Finding Redundancy in Web Mutation Operators</td>
<td>2017</td>
<td>IEEE</td>
<td>International Workshop on Mutation Analysis (Mutation)</td>
</tr>
<tr>
<td>[160]</td>
<td>Sahinoglu and Spafford</td>
<td>A Bayes Sequential Statistical Procedure for Approving Software Products</td>
<td>1990</td>
<td>Elsevier</td>
<td>IFIP Conference on Approving Software Products (ASP)</td>
</tr>
<tr>
<td>[162]</td>
<td>Schäfer and Zeller</td>
<td>Covering and Uncovering Equivalent Mutants based on Importance Sampling</td>
<td>2013</td>
<td>Wiley</td>
<td>Software Testing, Verification and Reliability</td>
</tr>
<tr>
<td>[163]</td>
<td>Srinath and Sam Lamin</td>
<td>Prioritizing Mutation Operators based on Importance Sampling</td>
<td>2010</td>
<td>IEEE</td>
<td>International Symposium on Software Reliability Engineering (ISSRE)</td>
</tr>
<tr>
<td>[164]</td>
<td>Steinmann and Thies</td>
<td>From Behaviour Preservation to Behaviour Modification: Constraint-Based Mutant Generation</td>
<td>2010</td>
<td>ACM</td>
<td>International Conference on Software Engineering (ICSE)</td>
</tr>
<tr>
<td>[167]</td>
<td>Tsengi, Suarez-Cal and de la Riva</td>
<td>Mutating database queries</td>
<td>2007</td>
<td>Elsevier</td>
<td>Information and Software Technology</td>
</tr>
<tr>
<td>[172]</td>
<td>Vinciotti, Malonado, Barbosa and Delamaro</td>
<td>Decreasing the Cost of Mutation Testing with Second-Order Mutants</td>
<td>2000</td>
<td>Wiley</td>
<td>Software Testing, Verification and Reliability</td>
</tr>
<tr>
<td>[173]</td>
<td>Vinciotti, Nakagawa, Malonado, Delamaro and Romero</td>
<td>Bayesian-Learning Based Guidelines to Determine Equivalent Mutants</td>
<td>2002</td>
<td>Wiley</td>
<td>Software Testing, Verification and Reliability</td>
</tr>
<tr>
<td>[174]</td>
<td>Vinciotti, Nakagawa, Malonado, Delamaro and Romero</td>
<td>Bayesian-Learning Based Guidelines to Determine Equivalent Mutants</td>
<td>2002</td>
<td>Wiley</td>
<td>Software Testing, Verification and Reliability</td>
</tr>
<tr>
<td>Id</td>
<td>Author</td>
<td>Title</td>
<td>Year</td>
<td>Publisher</td>
<td>Journal/Event</td>
</tr>
<tr>
<td>-----</td>
<td>-----------------------------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
<td>-----------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>185</td>
<td>Wang, Xiong, Shi, Zhang and Hao</td>
<td>Faster Mutation Analysis via Equivalence Modulo States</td>
<td>2017</td>
<td>ACM</td>
<td>International Symposium on Software Testing and Analysis (ISSTA)</td>
</tr>
<tr>
<td>186</td>
<td>Wedyan and Ghosh</td>
<td>On Generating Mutants for AspectJ Programs</td>
<td>2012</td>
<td>Elsevier</td>
<td>Information and Software Technology</td>
</tr>
<tr>
<td>189</td>
<td>Wong, Maldonado, Delamaro and Mathur</td>
<td>Constrained Mutation in C Programs</td>
<td>1994</td>
<td>Brazilian Computer Society</td>
<td>Brazilian Symposium on Software Engineering (SBES)</td>
</tr>
<tr>
<td>192</td>
<td>Wright, Kapfinger, and McMen</td>
<td>Efficient Mutation Analysis of Relational Database Structure Using Mutant Schemata and Parallelisation</td>
<td>2013</td>
<td>IEEE</td>
<td>International Workshop on Mutation Analysis (Mutation)</td>
</tr>
<tr>
<td>197</td>
<td>Zhang, Hou, Hu, Xie and Mei</td>
<td>Is Operator-Based Mutant Selection Superior to Random Mutant Selection?</td>
<td>2010a</td>
<td>ACM</td>
<td>International Conference on Software Engineering (ICSE)</td>
</tr>
<tr>
<td>200</td>
<td>Zhang, Xie, Zhang, Tillmann, de Halleux and Mei</td>
<td>Test Generation via Dynamic Symbolic Execution for Mutation Testing</td>
<td>2010b</td>
<td>IEEE</td>
<td>International Conference on Software Maintenance (ICSM)</td>
</tr>
<tr>
<td>199</td>
<td>Zhang, Marinov, Zhang and Khurshid</td>
<td>Regression Mutation Testing</td>
<td>2012</td>
<td>ACM</td>
<td>International Symposium on Software Testing and Analysis (ISSTA)</td>
</tr>
<tr>
<td>196</td>
<td>Zhang, Marinov and Khurshid</td>
<td>Faster Mutation Testing Inspired by Test Prioritization and Reduction</td>
<td>20136</td>
<td>ACM</td>
<td>International Symposium on Software Testing and Analysis (ISSTA)</td>
</tr>
<tr>
<td>198</td>
<td>Zhang, Golgoric, Marinov and Khurshid</td>
<td>Operator-Based and Random Mutant Selection: Better Together</td>
<td>2013a</td>
<td>IEEE</td>
<td>International Conference on Automated Software Engineering (ASE)</td>
</tr>
<tr>
<td>201</td>
<td>Zhu, Panchella and Zaidman</td>
<td>Speeding Up Mutation Testing via Data Compression and State Infection</td>
<td>2017</td>
<td>IEEE</td>
<td>International Workshop on Mutation Analysis (Mutation)</td>
</tr>
<tr>
<td>202</td>
<td>Zhu, Panchella and Zaidman</td>
<td>An Investigation of Compression Techniques to Speed up Mutation Testing</td>
<td>2018</td>
<td>IEEE</td>
<td>International Conference on Software Testing, Verification and Validation (ICST)</td>
</tr>
<tr>
<td>Updated / Extended Studies</td>
<td>Subsumed Studies</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ID</strong></td>
<td><strong>Author</strong></td>
<td><strong>Year</strong></td>
<td><strong>Title</strong></td>
<td><strong>ID</strong></td>
<td><strong>Author</strong></td>
</tr>
<tr>
<td>[150]</td>
<td>Papadakis and Malevris</td>
<td>2010</td>
<td>Towards Automating the Generation of Mutation Tests</td>
<td>[152]</td>
<td>Papadakis</td>
</tr>
</tbody>
</table>
Table A.15: Subsumed studies (2/2) (Id = Reference ID).

<table>
<thead>
<tr>
<th>Updated / Extended Studies</th>
<th>Subsumed Studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Author</td>
</tr>
<tr>
<td>156</td>
<td>Petrović and Ivanković</td>
</tr>
<tr>
<td>126</td>
<td>McMinn et al.</td>
</tr>
</tbody>
</table>