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

---

Dan Laurențiu, Șorecău Adrian, Șotropa Rafael Konstantinos

2025

ABSTRACT

Over the years, researchers have implemented numerous techniques to lower the cost of mutation testing. However, keeping track of all these methods and evaluating their cost-benefit tradeoffs is challenging, despite being crucial for practical applications. To address this, our paper conducted a systematic literature review, synthesizing existing research on mutation testing cost reduction. After reviewing 20 peer-reviewed studies, we identified six key cost-reduction objectives and six corresponding techniques.

Contents

Chapter 1

Introduction

Mutation testing is a testing criterion that has been extensively studied yet lightly used in the last 50 years. 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 or even identify redundancies in the codebase. 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 used to measure the quality of tests and other test criteria in many studies. Despite its effectiveness, several factors make it expensive and 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. For even small methods with a few dozen lines of code, tens of thousands of mutants can be generated. Some mutants are trivially killed (killed by all tests) and some are easily killed (killed by many tests), but some are more difficult and require detailed analysis by the tester. It is also very difficult to determine whether a mutant is equivalent or is simply difficult to kill. And not surprisingly, the most difficult-to-kill mutants lead to valuable test cases. These costs are a major obstacle to practical adoption of mutation testing. Researchers have invented many ways to reduce the cost, focusing on several goals. Specific goals include reducing the number of mutants , not creating certain mutants, speeding up mutant execution, automating test set generation, minimizing or prioritizing test sets, and automatically identifying equivalent mutants.

We analyzed 30 peer-reviewed, published, primary studies. We down-selected these to 20 studies by removing studies that were either extended or updated by later studies. We then characterized 6 main goals related to cost reduction, and then identified 6 cost reduction techniques.

The remainder of this review is organized as follows. Section 2 provides details of our

SLR protocol, goals, research question (RQ1), and the criteria and procedures we used to select and analyze the studies. Section 3 summarizes results from our search and gives answers to the research questions. Finally, the conclusions are presented in Section 4.

Chapter 2

Study Setup

2.1 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.

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

2.2 Study Selection Criteria

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.

We selected studies that first passed I3, and then either I1 or I2 $\rightarrow$ (I3 $\wedge$ (I1 $\vee$ I2)). We discarded studies that passed E1 or E2 $\rightarrow$ (E1 $\vee$ E2).

2.3 Selection Procedures

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

> ***("mutation testing" OR "mutation analysis" OR "mutant analysis") AND ("cost reduction" OR "selective mutation" OR "random mutation" OR "random mutants" OR "equivalent mutant" OR "equivalent mutation")***

The search string was submitted to the search engines of the following databases: *arXiv*, *Google Scholar*, *Semantic Scholar*, and *ResearchGate*.

Step 2 → Snowballing: The backward snowballing technique 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.

Chapter 3

Characterization of the Techniques

Our analysis resulted in the six primary goals described as follows. We also provide a complete list of references to studies that pursued each primary goal.

3.1 Introducing the main goals of the techniques

PG-1 → *Reducing the number of mutants*: The objective is to reduce the number of mutants that will be executed, preferably without reducing effectiveness.
[AA18][ESM99][MP12][MAH17][MAC13][IB14a][IB14b]

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.
[RH99][JO06][MK14][MP16][MK15]

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.
[LZ12][LC18]

PG-4 → *Reducing the number of test cases or 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.
[RG18]

PG-5 → *Avoiding the creation of certain mutants*: The objective is to define mutation operators or mutation generation algorithms to generate fewer mutants.
[FS10][RJ15]

PG-6 → *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.
[RAMF16][CH14][MP11]

Figure 3.1 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 automatically detecting equivalent mutants.

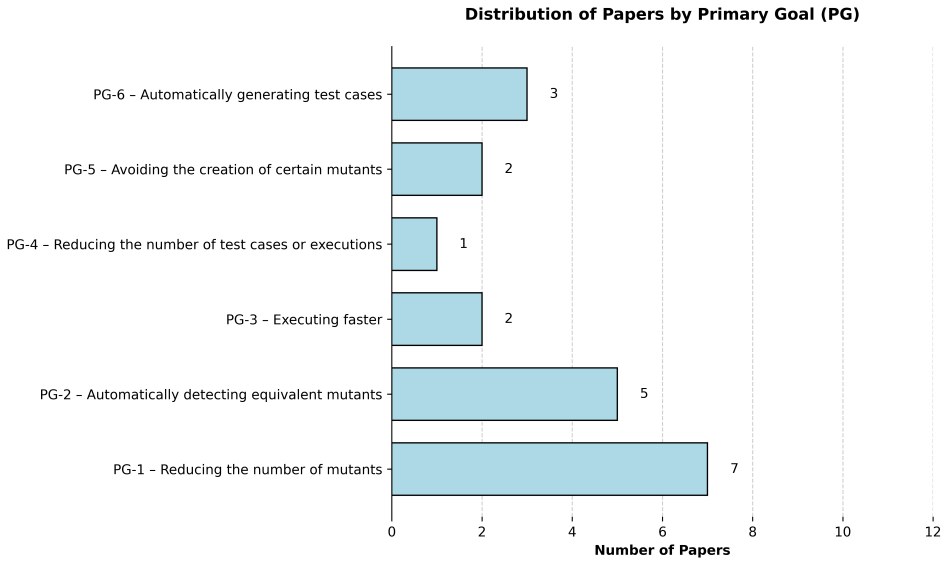

Figure 3.1: Distribution of Papers by Primary Goal (PG).

3.2 Introducing the techniques

This section presents a broad classification of cost reduction-related research according to 6 techniques. Some of the categories rely on mutation-specific research (higher order mutation), and others on classical software execution and analysis techniques (control-flow analysis).

T-1 → *Higher order mutation*: This technique combines two or more simple mutations to create a single complex mutant.

T-2 → *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.

T-3 → *Control-flow analysis*: This technique uses program control flow-related information, and focuses on execution characteristics to identify branches and commands that help determine which structures are most relevant to the generation and execution of mutants.

T-4 → *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.

T-5 → *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.

T-6 → *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.

Figure 3.2 shows the distribution of studies per technique. Some studies used two or more techniques.

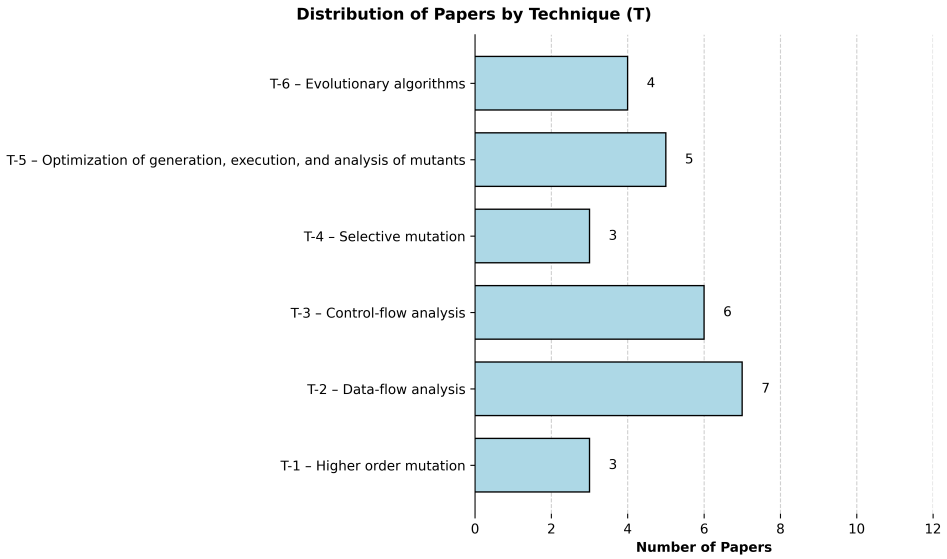

Figure 3.2: Distribution of Papers by Technique (T).

3.3 Summary of the Papers

The remainder of this section provides a more concise description of the selected studies, grouped by cost reduction goal.

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

"An Approach for the Generation of Higher Order Mutants Using Genetic Algorithms"[AA18] presents an approach for generating higher order mutants using a genetic algorithm to reduce the cost of mutation testing and increase its effectiveness, with the goal of producing subtle and harder to kill mutants and reducing the percentage of equivalent mutants. Kintis et al. used a genetic algorithm in order to generate higher order mutants. The study also uses a software tool called HOMJava, which implements that approach. The genetic algorithm was configured to run for 50 generations with a population size of 2000 mutants. The fitness of the mutants was calculated using a test suite of 800 test cases. The results show that the approach was actually able to produce subtle higher order mutants, with the fitness of mutants improving by almost 99% compared to first order mutants. The study also found that the percentage of equivalent mutants was around 4%.

Mustafa et al.[MAH17] discusses the application of mutation testing to highly-configurable systems, proposing techniques to reduce costs by utilizing static variability analysis and T-wise testing to limit the number of mutants. The authors propose to utilize static variability analysis to identify at which point in the source code a mutation operator should be applied. The study also uses static analysis, the TypeChef tool, to analyze the variable code and identify dependencies. The authors found that by applying variability analysis, they can cover pairwise dependencies with 6 different configurations instead of using all 256. Also, they only need 36 mutants to assess all feature interactions, which is approximately 0.37% of the previous 9728 mutants used.

"MESSI: Mutant Evaluation by Static Semantic Interpretation"[MP12] proposes a new approach to the mutant reduction problem in mutation testing using static analysis, called Mutant Evaluation by Static Semantic Interpretation (MESSI). This works by determining the usefulness of each mutant without reference to any particular test suite. The methods used in the study include applying random testing to 19 methods from six classes of the Java standard library, and using the Traffic Collision Avoidance System (TCAS) to evaluate the effectiveness of MESSI. The key findings are that MESSI can select mutants that are difficult to kill, but it does not always select the most difficult to kill mutants, and removing half the mutants has a significant effect on the mutation

score and probability of killing each mutant.

Bluemke et al.[IB14a] examines the effectiveness of reducing the number of mutants generated for mutation operators in Java programs. Mutants were generated using MuClipse and test cases using JUnit. The study also used two special tools: Mutants Remover and Console Output Analyzer, which were implemented especially for these experiments. The study found that the number of test cases generated automatically by CodePro was only 28.78%, and the number of mutants killed by automatically generated tests was 47.15%. Results are showing that reductions of 40-60% can achieve mean values of killed mutants greater than 95% while minimizing degradation in code coverage.

"Efficiency of Mutation Operators and Selective Mutation Strategies: an Empirical Study"[ESM99] compares the efficiency of different mutation operators and selective mutation strategies, finding that some strategies are more efficient than others in terms of cost and mutation score, with the most efficient strategies being those that balance score and cost. The study used the Mothra mutation system to generate mutants and the Godzilla system to generate test cases. A cost model was also used to estimate the cost of detecting equivalent mutants. The study found that the cost of detecting equivalent mutants is a significant part of the cost of mutation testing. The study also found that the efficiency of selective mutation strategies depends on the size of the program under test.

The authors of "Mutation Testing Strategies using Mutant Classification"[MP16] examine the effectiveness of mutant classification strategies in software testing, with results showing that these strategies can kill a significant percentage of mutants, but with varying effectiveness. The study uses mutant classification schemes based on coverage impact, which measures the difference in statement coverage between the original and mutant programs. The Proteum and gcov tools were used in order to produce and execute mutants, and to gather statement coverage information. The study finds that mutant classification strategies can be useful in isolating equivalent mutants, but they fail to kill some mutants, achieving a kill rate of approximately 95% of the introduced killable mutants. The examined strategies from this paper are 5-10% less effective than testing based on all mutants.

Cachia et al.[MAC13] proposed incremental mutation testing, a technique that leverages the iterative nature of agile development to reduce the computational expense of mutation testing, by limiting the scope of mutant generation to sections of code that have changed since the last mutation run. The study also proves that incremental mutation testing drastically reduces the number of mutants generated and the time required

to generate mutants and execute tests against them. The methods used involve applying incremental mutation testing to the differences between two versions of code, using static analysis to select which tests to execute, and collecting data about the total number of generated mutants and the number of killed mutants. The results show that incremental mutation testing significantly reduces the amount of mutants generated, resulting in faster execution time. Because of this, speed improvements of between 88% and 91% were observed.

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

"Employing second-order mutation for isolating first-order equivalent mutants"[MK14] presents a novel dynamic method based on higher-order mutation called I-EQM. Kintis et al. used the Javalanche mutation testing framework to generate and execute the mutants, and the tool's default execution settings were used for the application of the coverage impact technique. The I-EQM classifier identifies 81% of the live killable mutants and 46% of the total instances of equivalent mutants. The results of the study show that the I-EQM method achieves a correct classification of 81% of killable mutants with a precision of 71%.

"Using Program Slicing to Assist in the Detection of Equivalent Mutants"[RH99] explores the application of amorphous slicing and constraint solving to detect equivalent mutants in mutation testing, aiming to reduce the expense associated with human analysis of mutants. Constraint solving is used to generate test data to kill mutants and to prove some mutants equivalent. On a suite of eleven programs the system was able to detect 48% of equivalent mutants. This means that program slicing can simplify the mutant and help the tester find test cases that kill it.

"MEDIC: A Static Analysis Framework for Equivalent Mutant Identification"[MK15] evaluates the effectiveness and efficiency of MEDIC, a static analysis framework for equivalent mutant detection. The methods used in the study are the application of the MuJava mutation testing framework to specific classes of the Java test subjects. The study also used the WALA framework for program analysis. Kintis et al. found that MEDIC managed to detect 56% of the examined equivalent mutants in 125 seconds. MEDIC was also able to identify equivalent mutants in the JavaScript test subjects.

Offutt et al.focused on Class-Level Mutants of MuJava in "The Class-Level Mutants of MuJava"[JO06] . They explore mutation testing in practical software development. One important result is that class-level mutation operators yield far more equivalent mutants than traditional, statement-level, operators. This paper creates new Class-level mutation operations that do not introduce equivalent mutants. The new classes are: PCI (type cast operator insertion), PCD (type cast operator deletion), PCC (cast type change). Out of MuJava's total of 29 classes, 20 out of them resulted in zero mutants.

"Reductions of operators in Java mutation testing"[IB14b] examines the reduction of mutation operators in Java programs, with the goal to reduce the cost of testing while maintaining an acceptable mutation score. The study used MuClipse and CodePro plugins in the Eclipse environment, and two special tools: Mutants Remover and Console Output Analyzer. The study used the MuClipse set of mutation operators to generate mutants and measure the killed mutant factor. Bluemke et al. showed that selective mutation can decrease the number of mutants by 44% on average, and that the killed mutants factor was greater than 95% in 43 out of 48 cases, and even 100% in 33 cases.

3.3.3 Studies that Tried to Execute Faster (PG-3)

Chen et al.[LC18] explores the effectiveness of Regression Test Selection techniques in speeding up mutation testing, a methodology used to evaluate the quality of test suites. Both file-level static and dynamic RTS techniques can achieve precise and efficient mutation testing, resulting from this. The study uses various state-of-the-art dynamic and static, as well as file-level and method-level RTS techniques, and evaluates their effectiveness in speeding up mutation testing using the PIT tool. The study uses two RTS techniques: STARTS and Ekstazi. They also use an example program to illustrate the RTS-based regression mutation testing. The study finds that Ekstazi and STARTS can greatly reduce mutation testing costs, with test-level reduction of 78.30% and 73.31%.

"Regression Mutation Testing"[LZ12] presents a technique that should reduce the cost of mutation testing by leveraging program differences and utilizing mutation testing results from previous program versions. ReMT uses static analysis to check which results can be safely reused, and employs a mutation-specific test prioritization to speed up mutation testing even more. Zhang et al. found that ReMT can substantially reduce mutation testing costs, with reductions of more than 50% on the majority of the revisions and more than 90% on 12 of the 30 studied revisions.

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

"If You Can't Kill a Supermutant, You Have a Problem"[RG18] tries to reduce the resource requirements of mutation analysis by combining multiple mutants into supermutants and evaluating them together. The authors propose a method for computing the mutant kill matrix using supermutants, by generating a composite supermutant containing all the first-order mutations possible in the program, and then evaluating the supermutant against each test case. The study notes that the savings are also dependent on the strength of the test suite. Gopinath et al. finds that the supermutant technique can evaluate 5 mutants with a single test case, resulting in a total of 13,009 test evaluations.

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

Steimann et al.[FS10] discusses the transition from behavior preservation to behavior modification in mutant generation for programs. By generating mutants that are syntactically and semantically correct and likely exhibit non-equivalent behavior, this paper improves mutation analysis. This means that it reduces the number of irrelevant mutants generated. The methodology involves applying constraint-based refactoring techniques to generate mutants and then analyzing sample programs, written in java, and applying constraint rules to identify relevant mutants. This method is able to reduce by 87% the number of mutants generated.

Just et al.[RJ15] investigates the efficiency improvements of mutation analysis by eliminating redundant mutants in "Higher Accuracy and Lower Run Time: Efficient Mutation Analysis Using Non-Redundant Mutation Operators". The result was a significant reduction in the number of generated mutants and mutation analysis run time, with an overall decrease of 27% in the number of generated mutants and 22% in the run time for developer-written test suites. The methods used include the definition of non-redundant and subsumed mutations, the provision of a sufficient set of non-redundant mutations for the COR, UOI, and ROR mutation operators, and the empirical study that investigates how redundant mutants affect the efficiency and accuracy of mutation analysis. The study used the Major mutation framework to generate mutants and the EvoSuite

test generation tool to generate additional test suites. Eliminating redundant mutants decreases the total mutation analysis run time by more than 20%. The study also shows that the inclusion of redundant mutants misleadingly overestimates the mutation score.

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

"Automatic Mutation based Test Data Generation"[MP11] proposes a search-based test data generation approach for mutation testing, utilizing a novel dynamic execution scheme and a fitness function composed of four parts to guide the search process towards generating test cases that can kill mutants. This paper introduces an automated framework that produces test cases based on mutation testing. A hill climbing algorithm known as the alternating variable method (AVM) was used, and a fitness function composed of four parts, as well as the use of a dynamic approach level to guide the search process.

Filho et al.[RAMF16] evaluates the use of multi-objective evolutionary algorithms, specifically NSGA-II, SPEA2, and IBEA, to optimize test sets for software product lines. The authors found that these algorithms can produce diverse good solutions with reduced number of products. The approach uses three multi-objective algorithms: NSGA-II, SPEA2, and IBEA, and a new representation for the population, where an individual is a set of products. The approach also uses a fitness function that evaluates the quality of the solution. This approach can reduce the number of test cases, with a reduction of up to 16% of the number of products derived.

In Table 3.1, we have provided a list of all the papers processed along with their corresponding PG and T categories.

Cit	Title	PG	T
[AA18]	Approach for Generation of Higher Order Mutants	1	1, 6
[ESM99]	Efficiency of Mutation Operators and Selective Mutation	1	4
[LZ12]	Regression Mutation Testing	3	3, 5
[FS10]	From Behaviour Preservation to Behaviour Modification	5	5
[RJ15]	Higher Accuracy, Lower Run Time: Efficient Analysis	5	5
[RH99]	Program Slicing to Assist for Detection of Equivalent Mutants	2	2
[MP12]	MESSI: Mutant Evaluation by Static Semantic Interpretation	1	2, 3
[MAH17]	Efficient Mutation Testing in Configurable Systems	1	2
[JO06]	The Class-Level Mutants of MuJava	2	2, 3
[MAC13]	Towards Incremental Mutation Testing	1	5
[IB14a]	Reduction in Mutation Testing of Java Classes	1	4
[MK14]	Employing second-order mutation for isolating mutants	2	1, 2, 3
[RAMF16]	Multi-objective data generation approach	6	6
[MP16]	Mutation Testing Strategies using Mutant Classification	2	2, 3
[MK15]	MEDIC: A Framework for Equivalent Mutant Identification	2	2
[CH14]	Mutation-based Generation of Test Configurations	6	6
[LC18]	Speeding up Mutation Testing via Regression Test Selection	3	3, 5
[MP11]	Automatic Mutation based Test Data Generation	6	6
[IB14b]	Reductions of operators in Java mutation testing	1	4
[RG18]	If You Can't Kill a Supermutant, You Have a Problem	4	1

Table 3.1: Mutation Testing Studies and their Related Techniques and Primary Goals

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.

Chapter 4

Conclusion

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.

Future work: As with any SLR, it is unlikely that we found all primary studies. Even though several search strategies were used, well-known limitations of scientific writing and repositories mean no search can be perfect.

Bibliography

[AA18] Fadi Wedyan Anas Abuljadayel. An approach for the generation of higher order mutants using genetic algorithms. *International Journal of Intelligent Systems and Application*, 10(1), 2018.

[CH14] Yves Le Traon Christopher Henard, Mike Papadakis. Mutation-based generation of software product line test configurations. *Search-Based Software Engineering: 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26-29, 2014. Proceedings 6*, 68, 2014.

[ESM99] Leonardo Bottaci Elfurjani S. Mresa. Efficiency of mutation operators and selective mutation strategies: an empirical study. *Software Testing, Verification and Reliability*, 9(4), 1999.

[FS10] Andreas Thies Friedrich Steimann. From behaviour preservation to behaviour modification: Constraint-based mutant generation. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2010.

[IB14a] Karol Kulesza Ilona Bluemke. Reduction in mutation testing of java classes. *ICSOFT-EA 2014 - 9th International Conference on Software Engineering and Applications*, 2014.

[IB14b] Karol Kulesza Ilona Bluemke. Reductions of operators in java mutation testing. *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30–July 4, 2014, Brunów, Poland*, 2014.

[JO06] Yong-Rae Kwon Jeff Offutt, Yu-Seung Ma. The class-level mutants of mujava. 2006.

[LC18] Lingming Zhang Lingchao Chen. Speeding up mutation testing via regression test selection: An extensive study. *2018 IEEE 11th international conference on software testing, verification and validation (ICST)*, 2018.

[LZ12] Lu Zhang Lingming Zhang, Darko Marinov. Regression mutation testing. *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012.

[MAC13] Christian Colombo Mark Anthony Cachia, Mark Micallef. Towards incremental mutation testing. *Electronic Notes in Theoretical Computer Science*, 294, 2013.

[MAH17] Jacob Krüger Mustafa Al-Hajjaji. Efficient mutation testing in configurable systems. *2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE)*, 2017.

[MK14] Nicos Malevris Marinos Kintis, Mike Papadakis. Employing second-order mutation for isolating first-order equivalent mutants. *Journal of Software Testing, Verification and Reliability*, 25, 2014.

[MK15] Nicos Malevris Marinos Kintis. Medic: A static analysis framework for equivalent mutant identification. *Information and Software Technology*, 68, 2015.

[MP11] Nicos Malevris Mike Papadakis. Automatic mutation based test data generation. *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, 2011.

[MP12] Manuel Oriol Matthew Patrick. Messi: Mutant evaluation by static semantic interpretation. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012.

[MP16] Yves Le Traon Mike Papadakis. Mutation testing strategies using mutant classification. *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2016.

[RAMF16] Silvia R. Vergilio Rui A. Matnei Filho. A multi-objective test data generation approach for mutation testing of feature models. *Journal of Software Testing, Verification and Reliability*, 2016.

[RG18] Andreas Zeller Rahul Gopinath, Björn Mathis. If you can't kill a supermutant, you have a problem. *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2018.

[RH99] Sebastian Danicic Rob Hierons, Mark Harman. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4), 1999.

[RJ15] Franz Schweiggert Rene Just. Higher accuracy and lower run time: Efficient mutation analysis using non-redundant mutation operators. *Journal of Software Testing, Verification and Reliability*, 25, 2015.