

Mutation Testing: From Theory to Practice

Proefschrift voorgelegd tot het behalen van de
graad van Doctor in de Wetenschappen: Informatica
aan de Universiteit Antwerpen te verdedigen door:

Ali Parsai



Promoter
prof. dr. Serge Demeyer

Faculteit Wetenschappen
Departement Wiskunde-Informatica
Antwerpen, 2019

 Universiteit
Antwerpen

Cover Photo:

While normally peaceful creatures, when mutated, turtles are known to become fierce ninja warriors capable of defeating many a villain.

Mutation Testing: From Theory to Practice

Ali Parsai



Promoter:

prof. dr. Serge Demeyer

Proefschrift ingediend tot het behalen van de graad van
Doctor in de wetenschappen: Informatica

This dissertation has been approved by:

Promoter:

prof. dr. Serge Demeyer

Doctoral Jury:

prof. dr. Jeff Offutt

George Mason University, United States of America

prof. dr. Michail Papadakis

University of Luxembourg, Luxembourg

prof. dr. Guillermo Alberto Perez

University of Antwerp, Belgium

prof. dr. Jan Broeckhove

University of Antwerp, Belgium

prof. dr. Serge Demeyer

University of Antwerp, Belgium

*Dedicated to
Brother, for uncompromising support;
Grandfather, for the courage;
Mother, for everything.*

Acknowledgments

During the course of my PhD studies, I had the honor of getting to know so many amazing people who helped me along the way. First of all, I would like to thank my supervisor, Serge Demeyer, whose invaluable guidance helped me avoid the pitfalls in numerous occasions, and not only taught me most of what I know about research, but also in many ways he was a true sensei for a young samurai. I would like to thank my partner in crime, Alessandro Murgia from whom I learned a lot about how to write papers, play card games, and to always look on the bright side of life. I would like to thank my friends in LORE, in particular Diana Leyva Pernia, Gusher Laghari, Brent Van Bladel, Sten Vercammen, Henrique Rocha, and Mercy Njima who have helped me in many occasions. I would like to express my gratitude to the extended Ansymo family with whom I shared many good memories, Hans Vangheluwe, Simon Van Mierlo, Fons De Mey, Bart Meyers, Istvan David, and Claudio Gomes. During my research collaborations, I had the privilege of working with many wonderful people; Quinten Soetens and Filip Van Rysselberghe from OM Partners, Dennis Wagelaar from Health Connect/Corilus, Tars Joris from Inventive Designers, Yonni De Witte from Agfa HealthCare, Markus Borg from RISE, Sigrid Eldh from Ericsson, Joakim Brannstrom and Christoffer Nylen from SAAB, Mille Bostrom from SystemVerificaion, and many others whom I might have forgotten to mention. I am also grateful to my academic friends with whom I had many wonderful discussions; Mike Papadakis, Tom Mens, Alexandre Decan, Coen De Roover, and the whole of BENEVOL and A-TEST family. I am thankful of the help and support I received from the members of my PhD jury; Jeff Offutt and Mike Papadakis who accepted to be on the jury despite their busy schedule and gave me invaluable feedback. I would like to thank my friends Leonard Elezi, Asoh Frank, and Emad Heydari for their unwavering support through thick and thin, and lending me an ear whenever I needed to complain. Last but not least, I would like to thank my family; my mother, Fatemeh, my brother, Mohammad, and my father, Hossein who have supported me through this process, and made me happy when I was sad, and supported me when I needed it the most.

Ali Parsai
Antwerp, Belgium, August 2019

Abstract

The cost of software faults has increased from 59 billion USD in 2002 to 1.7 trillion USD in 2017. To alleviate this cost, the consensus among software engineers is to test as early and as often as possible. This, however, is not adopted by many software development teams. Most often, there are limited resources available for testing compared to the development of a product. Therefore, new techniques and methods are needed to improve testing quality in practice. Currently, most software companies rely on simple coverage metrics to assess the quality of their tests. Yet, the academic literature proposes the use of mutation testing to assess and improve the quality of software tests. Despite the promising results of mutation testing, it is not yet widely adopted in industry. We attribute this to three main problems: the performance overhead, lack of domain knowledge in tool providers, and lack of tool support. In this thesis, we address these three problems. Our results show that it is feasible to adapt the process of mutation testing based on industrial needs.

Nederlandstalige Samenvatting

De kosten veroorzaakt door softwarefouten zijn gestegen van 59 miljard USD in 2002 tot 1,700 miljard USD in 2017. Om deze kosten te verminderen is de consensus onder software engineers om zo vroeg en zo vaak mogelijk te testen. In de praktijk wordt dit echter niet door veel software ontwikkeling teams toegepast. Meestal zijn er minder middelen beschikbaar om het product te testen dan om het product te ontwikkelen. Daarom zijn er nieuwe technieken en methoden nodig om de kwaliteit van softwaretesten in de praktijk te verbeteren. Momenteel vertrouwen de meeste softwarebedrijven op eenvoudige coverage statistieken om de kwaliteit van hun testen te beoordelen. De academische literatuur stelt echter het gebruik van mutation testing voor om de kwaliteit van softwaretesten te beoordelen en te verbeteren. Ondanks de veelbelovende resultaten van mutation testing wordt het nog niet algemeen toegepast in de industrie. We schrijven dit voornamelijk toe aan drie problemen: de overhead van de uitvoering, gebrek aan domeinkennis en gebrek aan ondersteuning voor tools. In dit proefschrift behandelen we deze drie problemen. Onze resultaten tonen aan dat het mogelijk is om het mutation testing proces aan te passen aan industriële behoeften.

Publications and Presentations

Papers included in this thesis:

1. Ali Parsai, Alessandro Murgia, and Serge Demeyer. **LittleDarwin: A Feature-Rich and Extensible Mutation Testing Framework for Large and Complex Java Systems**. In *7th International Conference on Fundamentals of Software Engineering (FSEN 2017)*, 148–163. Tehran, Iran. April, 2017.
URL: https://doi.org/10.1007/978-3-319-68972-2_10.
2. Ali Parsai, Alessandro Murgia, and Serge Demeyer. **Evaluating random mutant selection at class-level in projects with non-adequate test suites**. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE 2016)*, 11:1–11:10. Limerick, Ireland. June, 2016.
URL: <https://doi.org/10.1145/2915970.2915992>.
3. Ali Parsai, Alessandro Murgia, and Serge Demeyer. **A Model to Estimate First-Order Mutation Coverage from Higher-Order Mutation Coverage**. In *2016 IEEE International Conference on Software Quality, Reliability, and Security (QRS 2016)*, 365–373. Vienna, Austria. August, 2016.
URL: <https://doi.org/10.1109/qrs.2016.48>.
4. Ali Parsai and Serge Demeyer. **Dynamic Mutant Subsumption Analysis Using LittleDarwin**. In *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing (A-TEST 2017)*, 1–4. Paderborn, Germany. September, 2017.
URL: <https://doi.org/10.1145/3121245.3121249>.
5. Ali Parsai, Serge Demeyer, and Seph De Busser. **C++11/14 Mutation Operators Based on Common Fault Patterns**. In *Proceedings of 2018 International Conference on Testing Software and Systems (ICTSS 2018)*, 102–118. Cadiz, Spain. October, 2018.
URL: https://doi.org/10.1007/978-3-319-99927-2_9.
6. Ali Parsai and Serge Demeyer. **Do Null-Type Mutation Operators Help Prevent Null-Type Faults?**. In *SOFSEM 2019: Theory and Practice of Computer Science (SofSem*

2019), 419–434.. January, 2019.

URL: https://doi.org/10.1007/978-3-030-10801-4_33.

7. Ali Parsai and Serge Demeyer. **Comparing Mutation Coverage Against Branch Coverage in an Industrial Setting (Under Review)**. In *International Journal on Software Tools for Technology Transfer (Under Review) (STTT)*, .. August, 2019.

URL: N/A.

Presentations and tutorials in the course of doctoral study:

1. 2016-01-26 **Adaptable Mutation Testing for Continuous Integration Environments** at HealthConnect (Vilvoorde, Belgium)
2. 2016-02-05 **Catching Difficult Bugs Early: Mutation Analysis Customized!** at Cha-Q Steering Board Meeting (Antwerp, Belgium)
3. 2016-12-05 **Improving the Test Suite Using Mutation Testing** at Cha-Q Public Tool Demo Event (Brussels, Belgium)
4. 2017-04-19 **Mutation Testing** at (UN)MANNED (Bruges, Belgium)
5. 2017-06-19 **Using Mutation Testing to Improve Test Quality** at Revamp² Plenary Meeting (Stockholm, Sweden)
6. 2017-10-24 **Software Engineering: Research and Practice** at ForEach (Antwerp, Belgium)
7. 2017-10-24 **Software Engineering: Research and Practice** at Capgemini (Diegem, Belgium)
8. 2018-02-06 **Test Quality** at Software Study Trip with Delaware and Barco (Lyon, France)
9. 2018-09-04 **Mutation Testing** at CoSys/ Ansymo Workshop (Antwerp, Belgium)
10. 2018-11-06 **Mutation Testing: Opportunities and Pitfalls** at Nederlandse Test Dag 2018 (Delft, The Netherlands)
11. 2019-02-06 **Test Automation as a Key to Success** at Software Study Trip with Delaware and Barco (Amsterdam, The Netherlands)
12. 2019-04-11 **Mutation Testing: Opportunities and Pitfalls** at Ericsson Testing Day (Stockholm, Sweden)

Contents

Acknowledgments	v
Publications and Presentations	xi
1 Introduction	1
1.1 Contributions	3
1.2 Origins of Chapters	4
2 Background	5
2.1 Invalid Mutants	5
2.2 Equivalent Mutants	6
2.3 Mutation Coverage	7
2.4 Mutation Operators	7
2.5 Mutant Sampling	8
2.6 First-Order Mutants and Higher-Order Mutants	9
2.7 Mutant Subsumption	9
2.8 Mutation Testing Tools	10
3 LittleDarwin: a Feature-Rich and Extensible Mutation Testing Framework for Large and Complex Java Systems	11
3.1 Introduction	12
3.2 Mutation Testing	13
3.3 Design and Implementation	15
3.4 Experiments	22
3.5 Conclusion	25
4 Evaluating Random Mutant Selection at Class-Level in Projects with Non-Adequate Test Suites	27
4.1 Introduction	28
4.2 Background	30
4.3 Case Study Setup	32
4.4 Results	35

4.5	Threats To Validity	41
4.6	Related Work	46
4.7	Conclusion and Future Work	47
5	A Model to Estimate First-Order Mutation Coverage from Higher-Order Mutation Coverage	49
5.1	Introduction	50
5.2	Background	52
5.3	Proposed Model	55
5.4	Case Study Design	56
5.5	Results and Discussion	59
5.6	Threats to Validity	66
5.7	Related Works	67
5.8	Conclusion	68
6	Dynamic Mutant Subsumption Analysis using LittleDarwin	69
6.1	Introduction	70
6.2	Background	71
6.3	State of the Art	72
6.4	Dynamic Mutant Subsumption Analysis with LittleDarwin	72
6.5	Conclusion	75
7	Do Null-Type Mutation Operators Help Prevent Null-Type Faults?	77
7.1	Introduction	78
7.2	Background and Related Work	79
7.3	Experimental Setup	81
7.4	Results and Discussion	82
7.5	Threats to Validity	90
7.6	Conclusion	91
8	C++11/14 Mutation Operators Based on Common Fault Patterns	93
8.1	Introduction	94
8.2	Background and Related Work	95
8.3	Study Design	99
8.4	Results	101
8.5	Conclusions and Future Work	110
9	Comparing Mutation Coverage Against Branch Coverage in an Industrial Setting	111
9.1	Introduction	112

9.2	Background	115
9.3	Tools Used in This Study	120
9.4	Case Study Design	123
9.5	Results and Discussion	128
9.6	Threats to Validity	142
9.7	Related Work	145
9.8	Conclusion and Future Work	145
10	Conclusion	149
	Bibliography	173

List of Figures

2.1	Example of an equivalent mutant in <code>proc2</code>	6
3.1	Data Flow Diagram for LittleDarwin Components	16
3.2	The Header of a LittleDarwin Mutant	17
3.3	LittleDarwin Project Report	18
3.4	Comparing mutation coverage from PITest, and LittleDarwin, and branch coverage from JaCoCo	24
4.1	Mutation coverage equation	33
4.2	An example of calculation of correlation	34
4.3	The distance between mutation coverage calculated from sampled set and full set for sampling rates from 1% to 100%	37
4.4	How the number of mutants per class affect the reduction of the sample size	39
4.5	How the number of classes with mutants affects the acceptable sampling rate	40
4.6	How mutation coverage affect the acceptable sampling rate	40
4.7	Pearson correlation between sampled and all mutants. The data reported in these figures is discrete. However for better visualization, the points are connected together.	42
4.7	(Continued) Pearson correlation between sampled and all mutants. The data reported in these figures is discrete. However for better visualization, the points are connected together.	43
4.8	Kendall correlation between sampled and all mutants. The data reported in these figures is discrete. However for better visualization, the points are connected together.	44
4.8	(Continued) Kendall correlation between sampled and all mutants. The data reported in these figures is discrete. However for better visualization, the points are connected together.	45
5.1	Example for emergence of equivalence due to the context	53
5.2	Example of the estimation provided by the model	56
5.3	Number of remaining classes after applying threshold t	58

5.4	R^2 between estimated results and empirical data for different thresholds for second-order mutation. The red line denotes the chosen threshold, and the green line shows the level of R^2 that the chosen threshold guaranties.	61
5.5	Empirical data and the estimated curve for $t = 10$. The red curve is the estimation provided by the model, and each blue dot represents a class. . .	62
5.6	R^2 between estimated results and empirical data for different thresholds. .	65
6.1	Dynamic Mutant Subsumption Component I/O	73
6.2	Dynamic Mutant Subsumption Graph for JTerminal	74
6.3	Mutants 72, 88, and 111 of JTerminal	74
7.1	The Surviving Non-Equivalent Null-Type Mutants	83
7.2	Two of the Equivalent Mutants Generated by Traditional Mutation Operators	84
7.3	One of the Equivalent Mutants Generated by Null-Type Mutation Operators	84
7.4	The Tests Written to Kill the Surviving Null-Type Mutants	85
7.5	Number of killed and survived mutants for each mutation operator	88
7.6	Ratio of Null-Type and Traditional Mutants in All, Killed, and Subsuming .	89
7.7	Ratio of Mutants by Each Mutation Operator in All, Killed, and Subsuming	90
8.1	Generated Mutants	109
8.2	Mutation Operator Scores	109
9.1	Example of an equivalent mutant in <code>proc2</code>	118
9.2	Agfa HealthCare Segmentation build components	124
9.3	Number of classes in each category for t values between 1 and 25 in the industrial case	128
9.4	Visual comparison of branch coverage (JaCoCo) versus mutation coverage (LittleDarwin) on the industrial case	132
9.5	Weak correlation between branch coverage (JaCoCo) and mutation coverage (LittleDarwin) on the industrial case	134
9.6	Branch coverage (JaCoCo) and mutation coverage (LittleDarwin) at class level for the open source cases	135
9.6	(Continued from previous page) Branch coverage (JaCoCo) and mutation coverage (LittleDarwin) at class level for the open source cases	136
9.7	Correlation between branch coverage (JaCoCo) and mutation coverage (LittleDarwin) on the open source cases	137
9.8	Linear relationship between number of mutants and total analysis time . .	140
9.9	Correlation of full mutation coverage against mutation coverage with weighted sampling	140
9.10	The distribution of the <i>weighted mutation sampling</i>	141

List of Tables

2.1	Reduced-set mutation operators (adapted from [1] ©ACM 2006)	8
3.1	LittleDarwin Mutation Operators	19
3.2	Null Type Faults and Their Corresponding Mutation Operators	19
3.3	Comparison of Features in Mutation Testing Tools	20
3.4	Comparison of Experimental Features in Mutation Testing Tools	21
3.5	Pilot Experiment Results	24
4.1	LittleDarwin mutation operators	31
4.2	Relevant statistics of the selected projects	32
4.3	acceptable sampling rate for uniform and weighted approaches	39
5.1	LittleDarwin mutation operators	54
5.2	Projects sorted by mutation coverage	58
5.3	Parameters of the estimated curve	60
5.4	Number of higher-order mutants in each category for each project ($t = 0$, all mutants included)	64
6.1	JTerminal Software Information	73
7.1	Null-Type Faults and Their Corresponding Mutation Operators	81
7.3	Mutation testing results for VideoStore	82
7.4	Mutants Generated by LittleDarwin and LittleDarwin-Null	87
8.1	Project Statistics	101
8.2	Results of FOR Operator	103
8.3	Results of LMB Operator	104
8.4	Results of FWD Operator	106
8.5	Results of INI Operator	108
9.1	Reduced-set mutation operators (adapted from [1] ©ACM 2006)	119
9.2	PITest mutation operators at the time of this study	122
9.3	LittleDarwin mutation operators	123

9.4	Descriptive statistics for cases under investigation	125
9.5	Categorization of differences between branch and mutation coverage . . .	127
9.6	Comparing branch coverage (JaCoCo) versus mutation coverage (LittleDarwin) on cases under study	133
9.7	Comparing branch coverage (JaCoCo) versus mutation coverage (LittleDarwin)—full coverage + weighted sampling)	141

Introduction

Software is everywhere. From airplanes to hairdryers, one would be hard-pressed to find an electronic device that is not in some form or shape using software. This has indeed had implications for the world we live in. In a world filled by software, the cost of software faults is significant. In 2002, NIST estimated the cost of software faults to be around 59 billion USD [2]. A recent survey from 2017 by consultancy firm Tricentis suggests that the cost of software faults in the year 2017 alone is more than 1.7 trillion USD [3]. It has been shown that software faults that are detected in the later phases of software development cost more to fix [4]. The problem of software faults is exacerbated by the rapid growth of software industry and a lack of focus on software quality in favor of fast product delivery [5].

The consensus among software engineers is to suggest testing software as early and as often as possible [6]. This is often hindered by a lack of allocated resources for testing during development phase [7]. While software companies adopt a more agile approach to software development [8], the companies that use agile practices such as test-driven development remain in minority [7, 9]. As a result, new techniques and methods need to be created and valorized to deal with technical debt, especially in case of software testing [10]. Practitioners willing to adopt testing improvement techniques produced in research labs can create a strategic advantage [11].

The discrepancy between the theory and practice on what is an appropriate method to measure and improve the quality of software testing is a symptom of this problem [7]. On the one hand, use of metrics such as branch and statement coverage to measure quality of software tests is common in industry [7, 12]. In addition, safety-critical software such as automotive and aeronautics software are encouraged to use modified condition-decision coverage by safety standards [13, 14]. On the other hand, the academic research

has studied these measures and shown their flaws extensively [15, 16, 17, 18, 19, 20]. To discover a software fault, (i) a test needs to execute the fault (*Reachability*), (ii) the fault should put the program in the wrong state (*Infection*), (iii) the wrong state should cause the output to be wrong (*Propagation*), and (iv) the test oracle needs to recognize the wrong output (*Reveal*) [21, 22]. A *good* test is designed to target all these steps, namely, create a failure that is propagated to the output, and is revealed by the test oracle. The prime reason why coverage metrics are unreliable as test quality metrics is that they ignore steps (ii) through (iv). Therefore, using coverage metrics, the developer gets no information about the quality of their test oracle. To alleviate this problem, the academic literature promotes the use of mutation testing.

Mutation testing is the process of deliberately injecting faults into a software system, and then verifying whether the tests can detect the injected fault [23]. It is used as a way to measure and improve the quality of software tests. In this method, the faulty version of the software is called a *mutant*. The fault detection capability of a test suite is determined by the percentage of the mutants that are *killed*, namely, whether the fault inside a mutant was revealed by a test. If all tests pass, the mutant has *survived*. Since mutation testing covers all the steps of discovering a fault, it is superior to simple coverage metrics [24, 25, 26]. This fact becomes more pronounced in safety-critical software, where quality of the tests is a major development concern. In addition, recent industrial experiments show that even a limited adoption of mutation testing is useful in improving the development workflow [27, 28].

Despite promising results, the theory is not yet fully adopted into practice. For instance, Gopinath et al. put forward that mutation testing “is generally not used by real-world developers with any frequency” [29]. A survey on software testing practices in Canada has shown that in 2013 industry was paying attention to mutation testing [7]. Large companies such as Google are also showing interest in using mutation testing to improve their testing quality [27, 28]. Yet, we are still far from wide-spread adoption [23]. The literature blames this mainly on the performance overhead, leading to the adagium—do fewer, do smarter, and do faster [30, 31]. However, based on our own collaborations with the industry in Belgium, we argue there are at least two more problems on the way to successful valorization of mutation testing. Based on these observations, in this thesis we address the following problems:

- **Performance Problem:** Mutation testing is computationally intensive. For example, during one of our case studies, we encountered a 7 days run time of our tool for a project of moderate size (38 KLOC).
- **Fault Model Problem:** Mutation testing requires a fault model that represents the common faults in its target context. For example, when collaborating with a company that used modern C++ language constructs, we noticed that default set of

mutation operators omit mutating C++11/14 constructs, and thus, neglects an important part of their software.

- **Tool Problem:** Mutation testing tools cannot handle the complexity of industrial software. For example, in one of our case studies, we noticed that the complex structure of the build environment and its dependence on OSGI and separate project structures for testing does not allow us to use any of the mutation testing tools that were available at the time.

Indeed, there are more problems to be solved by researchers other than the aforementioned three. The equivalent mutant problem—where the generated mutant is semantically the same as the program—is a well-known example. While a major part of the literature consists of attempts to solve this problem, we did not find it an immediate roadblock for industrial adoption of mutation testing. Some industrial studies even mention the usefulness of such mutants [28]. Therefore, in this thesis we set the scope to solve only these three problems.

In Chapters 4 to 6 we address the performance problem. In Chapters 7 and 8 we address the fault model problem. Finally, in Chapters 3 and 9 we address the tool problem.

1.1 CONTRIBUTIONS

The main contributions of this thesis are as follows.

- In Chapter 3, we create and evaluate LittleDarwin as a mutation testing tool that can be used in a complex industrial environment.
- In Chapter 4, we evaluate random mutant selection as a method of reducing the performance cost of mutation testing on industrial-grade open-source software systems.
- In Chapter 5, we propose a model to estimate mutation coverage from higher-order mutation coverage in order to facilitate the use of higher-order mutation testing as a method of reducing performance cost of mutation testing.
- In Chapter 6, we facilitate dynamic subsumption analysis as a way to identify and remove redundant mutants by implementing it in LittleDarwin.
- In Chapter 7, we propose 4 new mutation operators to address the null-type problem in Java based on feedback from our industrial partners and evaluate them by performing a case study on industrial-grade open-source systems.
- In Chapter 8, we propose and evaluate 4 mutation operators for C++11/14 language features based on common fault patterns described by domain experts.

- In Chapter 9, we compare mutation coverage with branch and statement coverage in an industrial setting in collaboration with an industrial partner.

1.2 ORIGINS OF CHAPTERS

Chapters 3 to 8 are peer-reviewed and published. Chapter 9 is currently under peer-review.

CHAPTER 3 was published in the *7th International Conference on Fundamentals of Software Engineering (FSEN 2017)* [32].

CHAPTER 4 was published in the *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE 2016)* [33].

CHAPTER 5 was published in the *2016 IEEE International Conference on Software Quality, Reliability, and Security (QRS 2016)* [34].

CHAPTER 6 was published in the *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing (A-TEST 2017)* [35].

CHAPTER 7 was published in the *SOFSEM 2019: Theory and Practice of Computer Science (SofSem 2019)* [36].

CHAPTER 8 was published in the *Proceedings of 2018 International Conference on Testing Software and Systems (ICTSS 2018)* [37].

CHAPTER 9 is submitted to *International Journal on Software Tools for Technology Transfer (Under Review) (STTT)*.

Background

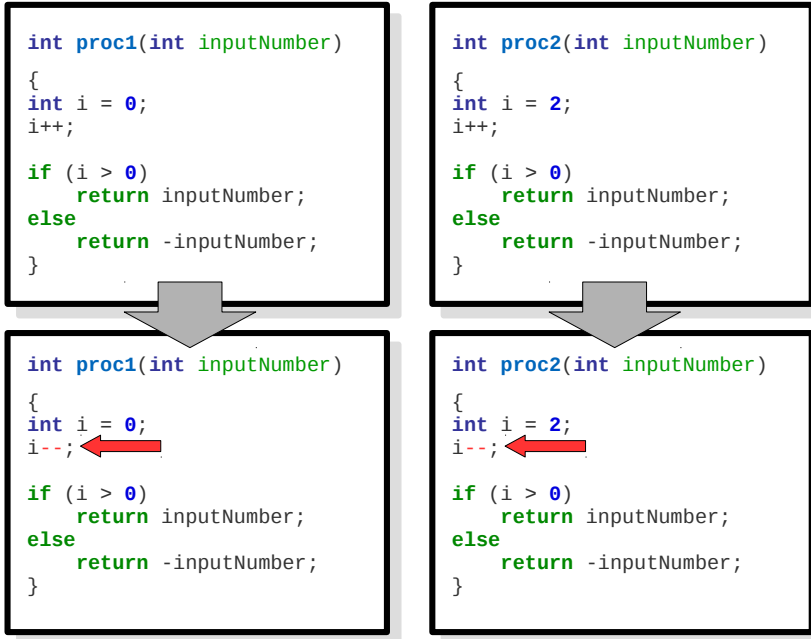
In this chapter, we provide the background information about mutation testing that is necessary to understand the rest of this thesis.

Mutation testing is the process of injecting faults into a software system and then verifying whether the test suite indeed fails (i.e. detects the injected fault). The idea of mutation testing was first mentioned in a class paper by Lipton (as reported by Offutt et al. in [30]), and later developed by DeMillo, Lipton and Sayward [38]. The first implementation of a mutation testing tool was done by Timothy Budd in 1980 [39].

Mutation testing induces the following steps on the test process. It starts with a *green* test suite — a test suite in which all the tests pass. First, a faulty version of the software is created by introducing faults into the system (*Mutation*). This is done by applying a known transformation (*Mutation Operator*) on a certain part of the code. After generating the faulty version of the software (*Mutant*), it is passed on to the test suite. If there is an error or failure during the execution of the test suite the mutant is marked as killed (*Killed Mutant*). If all tests pass, it means that the test suite could not catch the fault and the mutant has survived (*Survived Mutant*).

2.1 INVALID MUTANTS

In the process of generation of mutants, sometimes a mutant is not compilable. Such mutants are called *invalid mutants*. Given the fact that typical mutation testing tools do not attempt to compile the code entirely, it is possible that mutants are created that adhere to the syntax of a language, but cannot be compiled. For example, in case of concatenation of two string variables using “+” operator, changing this operator to “-” leads to generation of an invalid mutant. While most invalid mutants can be avoided at mutant generation

Figure 2.1: Example of an equivalent mutant in `proc2`

time, some are difficult to filter out without having the facilities of a compiler.

2.2 EQUIVALENT MUTANTS

If the output of a mutant for all possible inputs is the same as the original program, it is called an *equivalent mutant*. It is not possible to create a test case that passes for the original program and fails for an equivalent mutant, because the equivalent mutant has the same semantics as the original program. This makes the creation of equivalent mutants undesirable, since the time that the developer wastes on an equivalent mutant does not result in the improvement of the test suite. Equivalent mutants have a significant impact on the accuracy of the mutation coverage [40]. Unfortunately, equivalent mutants are not easy to detect because they depend on the context of the program itself [41]. For example, in Figure 2.1, `--` replacing `++` in `proc1` changes the output for any input other than 0, while the same mutant in `proc2` does not. Indeed, the preceding line `i++` ensures that the condition `i > 0` is always met for $i \geq 1$. The mutant can be killed in `proc1`, because $i = 0$, however in `proc2` the mutant is undetectable by any test because of $i = 2$.

As for filtering the equivalent mutants, there are no tools available that automatically detect and remove all equivalent mutants. In general, detection of equivalent mutants is an undecidable problem [42]. Manual inspection of all mutants is the only way of filtering

all equivalent mutants, which is impractical due to the amount of work it needs. Therefore, the common practice within today's state-of-the-art is to take precautions to remove as many equivalent mutants as possible (e.g. using Trivial Compiler Equivalence [43]), and accept equivalent mutants as a threat to validity.

2.3 MUTATION COVERAGE

Mutation testing allows software engineers to monitor the fault detection capability of a test suite by means of *Mutation Coverage* (see Equation 2.1). A test suite is said to achieve *full mutation test adequacy* whenever it can kill all of the non-equivalent mutants, thus reaches a mutation coverage of 100%. Such test suites are called *mutation-adequate test suites*.

$$\textit{Mutation Coverage} = \frac{\textit{Number of killed mutants}}{\textit{Number of all non-equivalent mutants}} \quad (2.1)$$

Mutation coverage is often declared as a *stopping criterion* for writing (unit) tests — the next level of testing can only start when mutation coverage exceeds a given threshold [44, 45]. This is especially useful when tests are generated automatically [46, 47].

2.4 MUTATION OPERATORS

A mutation operator is a known transformation which creates a faulty version by introducing a single change. The first set of the mutation operators designed were reported in King et al. [48]. These operators which work on very basic entities were introduced in the tool Mothra which was designed to mutate FORTRAN77 programming language. In 1996, Offutt et al. determined that a selection of few mutation operators are enough to produce similarly capable test suites with a four-fold reduction of the number of mutants [49]. This reduced set of operators shown in Table 2.1 remained more or less intact in all subsequent research papers.

With the popularity of the object-oriented programming paradigm, there was a need to design new mutation operators to simulate the faults that occur in this kind of programs. Several studies proposed new mutation operators [50, 51], and some of them were designed to prove the usefulness of object-oriented operators [52, 53]. Ahmed et al. did a complete survey on this subject [54].

During the past decade, the academic focus was on creating new mutation operators for special purposes such as targeting certain security problems [55, 56] or language specific mutation operators [36, 37, 57, 58, 59]. These mutation operators, even though im-

Table 2.1: Reduced-set mutation operators (adapted from [1] ©ACM 2006)

Operator	Description
AOR	Arithmetic Operator Replacement
AOD	Arithmetic Operator Deletion
AOI	Arithmetic Operator Insertion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COD	Conditional Operator Deletion
COI	Conditional Operator Insertion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOD	Logical Operator Deletion
LOI	Logical Operator Insertion
ASR	Assignment Operator Replacement

portant in their own context, do not relegate into the general concept of mutation testing. The traditional mutation operators are by far the most often implemented [23]. One reason for this is that using more mutation operators produces more mutants; which makes the procedure longer to finish, and as a result, less practical. The reduced set of operators mentioned in Table 2.1 provides a smaller set which produces results with enough detail for any practical purpose, even though the confidence in such results are slightly less than those retrieved by using additional mutation operators.

2.5 MUTANT SAMPLING

To make mutation testing practically applicable, it is important to reduce the time needed — do fewer, do smarter, and do faster [30]. “Do fewer” is achieved by *mutant sampling*: randomly selecting a sample set of mutants instead of processing all of them. This idea was first proposed by Acree [60] and Budd [39] in their PhD theses. Since then, there were many studies confirming the effectiveness of this approach: the performance gain is significant yet reveals the same weaknesses [48, 61, 62, 63]. The random mutant selection can be performed uniformly, meaning that each mutant has the same chance of being selected. Otherwise, the random mutant selection can be enhanced by using heuristics based on the source code.

The percentage of mutants that are selected determines the *sampling rate* for random mutant selection. Using a fixed sampling rate is common in literature [63, 64, 65]. However, it is possible to use a weight factor to optimize the sampling rate according to various parameters such as the number of mutants per class. This is called *weighted* mutant sampling [33]. It is also possible to determine the sampling rate dynamically while performing mutation testing. A method resembling the latter was proposed by Sahinoglu and Spafford to randomly select the mutants until the sample size becomes statistically

appropriate [66]. They concluded that their model achieves better results due to its self-adjusting nature [31].

There is one other factor besides the sampling rate that needs to be considered when sampling; the total amount of time that is practically viable. Unfortunately, in the current literature we did not find any concrete targets. Therefore, we set our own target based on a hypothetical scenario of an agile team running the whole mutation testing once every week during the weekend. In this scenario, the team works from Monday at 8am till Friday at 6pm, which leaves the whole weekend (thus 62 hours) to perform the analysis.

2.6 FIRST-ORDER MUTANTS AND HIGHER-ORDER MUTANTS

First-order mutants are the mutants generated by applying a mutation operator on the source code only once. By applying mutation operators more than once we obtain higher-order mutants. Higher-order mutants can also be described as a combination of several first-order mutants. Jia et al. [67] introduced the concept of higher-order mutation testing and discussed the relation between higher-order mutants and first-order mutants. They divided the higher-order mutants into four categories based on the observed coupling effect [68]: *Expected*, *Worst*, *Fault Shift*, and *Fault Mask*.

2.7 MUTANT SUBSUMPTION

Mutant subsumption is defined as the relationship between two mutants A and B in which A subsumes B if and only if the set of inputs that kill A is guaranteed to kill B [69]. The subsumption relationship for faults has been defined by Kuhn in 1999 [70], but its use for mutation testing has been popularized by Jia et al. for creating hard to kill higher-order mutants [67]. Later on, Ammann et al. tackled the theoretical side of mutant subsumption [71]. In their paper, Ammann et al. define *dynamic* mutant subsumption, which redefines the relationship using test cases. Mutant A dynamically subsumes Mutant B if and only if (i) A is killed, and (ii) every test that kills A also kills B. Kurtz et al. [69] use the notion of dynamic mutant subsumption graph (DMSG) to visualize the concept of dynamic mutant subsumption. Each node in a DMSG represents a set of all mutants that are mutually subsuming. Edges in a DMSG represent the dynamic subsumption relationship between the nodes. They introduce the concept of static mutant subsumption graph, which is a result of determining the subsumption relationship between mutants using static analysis techniques. The main purpose behind the use of mutant subsumption is to reliably detect redundant mutants, which create multiple threats to the validity of mutation testing [72]. This is often done by determining the dynamic subsumption relationship among a set of mutants, and keeping only those that are not subsumed by any other mutant.

2.8 MUTATION TESTING TOOLS

Javalanche is a mutation testing framework for Java programs that attempts to be efficient, and not produce equivalent mutants [73]. It uses byte code manipulation in order to speed up the process of mutation testing. Javalanche has been used in numerous studies in the past (e.g. [47, 74]).

PITest is a state-of-the-art mutation testing system for Java, designed to be fast and scalable [75]. PITest is the de facto standard for mutation testing within Java, and it is used as a baseline in mutation testing research (e.g. [76, 77]).

LittleDarwin is a mutation testing tool designed to work out of the box with complicated industrial build systems. For this, it has a loose coupling with the test infrastructure, instead relying on the build system to run the test suite. LittleDarwin has been used in several studies, and is capable of performing mutation testing on complicated software systems [33, 34, 78]. For more information about LittleDarwin please refer to Chapter 3.

LittleDarwin: a Feature-Rich and Extensible Mutation Testing Framework for Large and Complex Java Systems



LittleDarwin: A Feature-Rich and Extensible Mutation Testing Framework for Large and Complex Java Systems

Ali Parsai, Alessandro Murgia, and Serge Demeyer

In *7th International Conference on Fundamentals of Software Engineering (FSEN 2017)*, 148–163. Tehran, Iran. April, 2017.

URL: https://doi.org/10.1007/978-3-319-68972-2_10.

This chapter was originally published in the *7th International Conference on Fundamentals of Software Engineering (FSEN 2017)*.

CONTEXT

This chapter targets the third of our three identified problems, the tool problem. In particular, it describes the main tool used during the course of the PhD studies. LittleDarwin is built with the aim of easy deployment in complicated industrial environments. It has been updated with new features to allow for industrial case studies on mutation testing concepts.

We added a sanity check experiment and added information about incremental analysis to this chapter compared to the originally published paper.

ABSTRACT

Mutation testing is a well-studied method for increasing the quality of a test suite. We designed LittleDarwin as a mutation testing framework able to cope with large and complex Java software systems, while still being easily extensible with new experimental components. LittleDarwin addresses two existing problems in the domain of mutation testing: having a tool able to work within an industrial setting, and yet, be open to extension for cutting edge techniques provided by academia. LittleDarwin already offers higher-order mutation, null type mutants, mutant sampling, manual mutation, and mutant subsumption analysis. There is no tool today available with all these features that is able to work with typical industrial software systems.

3.1 INTRODUCTION

Along with the popularity of agile methods in recent times came an emphasis on test-driven development and continuous integration [79, 80]. This implies that developers are interested in testing their software components early and often [6]. Therefore, the quality of the test suite is an important factor during the evolution of the software. One of the extensively studied methods to improve the quality of a test suite is mutation testing [38].

Mutation testing was first proposed by DeMillo, Lipton, and Sayward to measure the quality of a test suite by assessing its fault detection capabilities [38]. Mutation testing has been shown to simulate faults realistically [24, 81]. This is because the faults introduced by each mutant are modeled after common mistakes developers make [31]. Mutation testing is demonstrated to be a more powerful coverage criteria in comparison with data-flow, statement, and branch coverage [25, 82].

Recent trends in scientific literature indicate a surge in popularity of this technique, along with an increased usage of real projects as the subjects of scientific experiments [31]. In literature, topics such as creating more robust mutants using higher-order mutation [68, 83, 84, 85], reducing redundancy among mutants using mutant subsumption [71, 72, 86], and reducing the number of mutants using mutant selection [65, 87, 88] are gaining popularity. Despite its benefits, the idea of mutation testing is not widely used in industry. Consequently, mutation testing research stays behind since it lacks fundamental experiments on industrial software systems. We believe that, beyond the computationally expensive nature of mutation testing [30], the reluctance of industry can stem from the shortage of mutation testing tools that can both (i) work on large and complex systems, and (ii) incorporate new and upcoming techniques as an experimental framework.

In this paper, we try to fill this gap by introducing LittleDarwin. LittleDarwin is designed as a mutation testing framework aiming to target large and complex systems. The design decisions are geared towards a simple architecture that allows the addition of new experimental components, and fast prototyping. In its current version, LittleDarwin fa-

facilitates experimentation on higher-order mutation, null type mutants, mutant sampling, manual mutation, and mutant subsumption analysis. LittleDarwin has been used for experimentation on several large and complex open source and industrial projects [33, 34, 89].

The rest of the paper is structured as follows. We provide background information about mutation testing in Section 3.2. We explain the design and the implementation of our tool in Section 3.3, and summarize the experiments that have been performed using our tool in Section 3.4. We conclude the paper in Section 3.5.

3.2 MUTATION TESTING

Mutation testing¹ is the process of injecting faults into a software system to verify whether the test suite detects the injected fault. Mutation testing starts with a *green* test suite — a test suite in which all the tests pass. First, a faulty version of the software is created by introducing faults into the system (*Mutation*). This is done by applying a known transformation (*Mutation Operator*) on a certain part of the code. After generating the faulty version of the software (*Mutant*), it is passed onto the test suite. If there is an error or failure during the execution of the test suite, the mutant is marked as killed (*Killed Mutant*). If all tests pass, it means that the test suite could not catch the fault, and the mutant has survived (*Survived Mutant*) [31].

Mutation Operators. A mutation operator is a transformation which introduces a single syntactic change into its input. The first set of mutation operators were reported in King et al. [48]. These mutation operators work on essential syntactic entities of the programming language such as arithmetic, logical, and relational operators. They were introduced in the tool Mothra which was designed to mutate the programming language FORTRAN77. In 1996, Offutt et al. determined that a selection of few mutation operators is enough to produce similarly capable test suites with a four-fold reduction of the number of mutants [49]. This reduced-set of operators remained more or less intact in all subsequent research papers. With the advent of object-oriented programming languages, new mutation operators were proposed to cope with the specifics of this programming paradigm [50, 90].

Equivalent Mutants. If the output of a mutant for all possible input values is the same as the original program, it is called an *equivalent mutant*. It is not possible to create a test case that passes for the original program and fails for an equivalent mutant, because the equivalent mutant is indistinguishable from the original program. This makes the creation of equivalent mutants undesirable, and leads to false positives during mutation

¹The idea of mutation testing was first mentioned by Lipton, and later developed by DeMillo, Lipton and Sayward [38]. The first implementation of a mutation testing tool was done by Timothy Budd in 1980 [39].

testing. In general, detection of equivalent mutants is undecidable due to the halting problem [42]. Manual inspection of all mutants is the only way of filtering all equivalent mutants, which is impractical in real projects due to the amount of work it requires. Therefore, the common practice within today's state-of-the-art is to take precautions to generate as few equivalent mutants as possible, and accept equivalent mutants as a threat to validity (accepting a false positive is less costly than removing a true positive by mistake [91]).

Mutation Coverage. Mutation testing allows software engineers to monitor the fault detection capability of a test suite by means of mutation coverage (see Equation 3.1) [31]. A test suite is said to achieve *full mutation test adequacy* whenever it can kill all the non-equivalent mutants, thus reaching a mutation coverage of 100%. Such test suite is called a *mutation-adequate test suite*.

$$\text{Mutation Coverage} = \frac{\text{Number of killed mutants}}{\text{Number of all non-equivalent mutants}} \quad (3.1)$$

Higher-Order Mutants. First-order mutants are the mutants generated by applying a mutation operator on the source code only once. By applying mutation operators more than once we obtain higher-order mutants. Higher-order mutants can also be described as a combination of several first-order mutants. Jia et al. introduced the concept of higher-order mutation testing and discussed the relation between higher-order mutants and first-order mutants [67].

Mutant Subsumption. Mutant subsumption is defined as the relationship between two mutants A and B in which A subsumes B if and only if the set of inputs that kill A is guaranteed to kill B [69]. The subsumption relationship for faults has been defined by Kuhn in 1999 [70], but its use for mutation testing has been popularized by Jia et al. for creating hard to kill higher-order mutants [67]. Later on, Ammann et al. tackled the theoretical side of mutant subsumption [71]. In their paper, Ammann et al. define *dynamic* mutant subsumption, which redefines the relationship using test cases. Mutant A dynamically subsumes Mutant B if and only if (i) A is killed, and (ii) every test that kills A also kills B. The main purpose behind the use of mutant subsumption is to reliably detect redundant mutants, which create multiple threats to the validity of mutation testing [72]. This is often done by determining the dynamic subsumption relationship among a set of mutants, and keeping only those that are not subsumed by any other mutant.

Mutant Sampling. To make mutation testing practical, it is important to reduce its execution time. One way to achieve this is to reduce the number of mutants. A simple approach to mutant reduction is to randomly select a set of mutants. This idea was first proposed by Acree [60] and Budd [39] in their PhD theses. To perform random mutant sampling, no extra information regarding the context of the mutants is needed. This

makes the implementation of this technique in mutation testing tools easier. Because of this, and the simplicity of random mutant sampling, its performance overhead is negligible. Random mutant sampling can be performed uniformly, meaning that each mutant has the same chance of being selected. Otherwise, random mutant sampling can be enhanced by using heuristics based on the source code. The percentage of mutants that are selected determines the *sampling rate* for random mutant sampling.

3.3 DESIGN AND IMPLEMENTATION

In this section, we discuss the implementation details of LittleDarwin, and provide information on our design decisions.

3.3.1 Algorithm

LittleDarwin is designed with simplicity in mind, in order to increase the flexibility of the tool. To this effect, it mutates the Java source code rather than the byte code in order to defer the responsibility of compiling and executing the code to the build system. This allows LittleDarwin to remain as flexible as possible regarding the complexities stemming from the build and test structures of the target software. The procedure is divided into two phases: *Mutation Phase* (Algorithm 1), and *Test Execution Phase* (Algorithm 2).

Mutation Phase. In this phase, the tool creates the mutants for each source file. LittleDarwin first searches for all source files contained in the path given as input, and adds them to the processing queue. Then, it selects an unprocessed source file from the queue, parses it, applies all the mutation operators, and saves all the generated mutants.

Algorithm 1: Mutation Phase

Input : Java source files

Output : Mutated Java source files

```

1 queue ← all Java source files;
2 while queue ≠ ∅ do
3   srcFile ← queue.pop();
4   mutants[srcFile] ← mutate(srcFile);
5 return mutants;
```

Test Execution Phase. In this phase, the tool executes the test suite for each mutant. First the build system is executed without any change to ensure that the test suite runs “green”. Then, a source file along with its mutants are read from the database, and the output of the build system is recorded for each mutant. If the build system fails (exits with non-zero status) or times out, the mutant is categorized as killed. If the build system

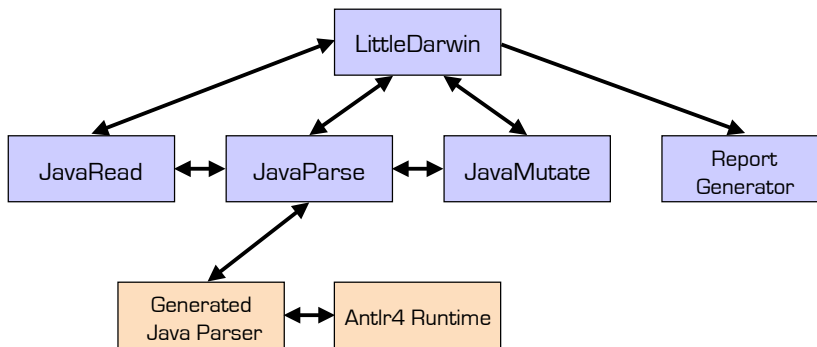


Figure 3.1: Data Flow Diagram for LittleDarwin Components

is successful (exits with zero status), the mutant is categorized as survived. Finally, a report is generated for each source file, and an overall report is generated for the project (see Figure 3.3 for an example of this).

Algorithm 2: Test Execution Phase

Input : Mutated Java source files

Output : Mutation Testing Report

```

1 if executeTestSuite() is successful then
2   foreach srcFile do
3     queue ← mutants[srcFile];
4     backup(srcFile);
5     while queue ≠ ∅ do
6       mutantFile ← queue.pop();
7       replace(srcFile,mutantFile);
8       result[mutantFile] ← executeTestSuite();
9     restore(srcFile);
10    Generate report for srcFile;
11  Generate overall report;
12 return reports;
    
```

3.3.2 Components

The data flow diagram of the main internal components of LittleDarwin is shown in Figure 3.1. The following is an explanation of each main component:

JavaRead. This component provides methods to perform input/output operations on Java files. LittleDarwin uses this component to read the source files, and write the

```

/* LittleDarwin generated mutant
  mutant type: relationalOperatorReplacement
  ----> before: daysRented  <= 0
  ----> after: daysRented  >0
  ----> line number in original file: 35
  ----> mutated nodes: 66
*/

```

Figure 3.2: The Header of a LittleDarwin Mutant

mutants back to disk.

JavaParse. This component parses Java files into an abstract syntax tree. This is necessary to produce valid and compilable mutants. To implement this functionality, an Antlr4² Java 8 grammar is used along with a customized version of Antlr4 runtime. Beside providing the parser, this component also provides the functionality to pretty print the modified tree back to a Java file.

JavaMutate. This component manipulates the abstract syntax tree (AST) created by the parser. Subsection 3.3.3 explains the mutation operators of LittleDarwin in detail. The currently implemented mutation operators search the provided AST for mutable nodes matching the predefined patterns (for example, *AOR-B* looks for all binary arithmetic operator nodes that do not contain a string as an operand), and they perform the mutation on the tree itself. This gives the developer flexibility in creating new complicated mutation operators. Even if a mutation operator introduces a fault that needs to change several statements at once, and depends on the context of the statements, it can be implemented using a complicated search pattern on the AST. The mutation operators are designed to exclude mutations that would lead to compilation errors. However, not all of these cases can be detected using an AST (e.g. *AOR-B* on two variables that contain strings). Handling of such cases are therefore left for the post-processing unit that filters such mutants based on the output of the Java compiler. In order to preserve the maximum amount of information for post-processing purposes, for each mutant a commented header is created. This header contains the following information: (i) the mutation operator that created the mutant, (ii) the mutated statement before and after the mutation, (iii) the line number of the mutated statement in the original source file, and (iv) the id number of the mutated node(s). An example is shown in Figure 3.2.

Report Generator. This component generates HTML reports for each file. These reports contain all the generated mutants and the output of the build system after the execution of each mutant. In the end, an overall report is generated for the whole project (Figure 3.3).

²<http://www.antlr.org/>

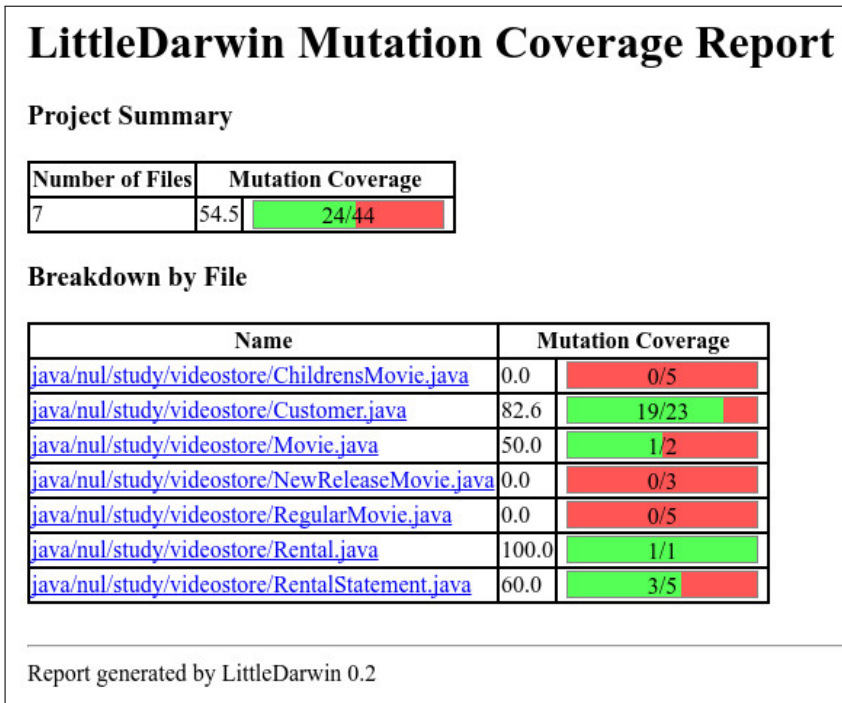


Figure 3.3: LittleDarwin Project Report

3.3.3 Mutation Operators of LittleDarwin

There are 9 default mutation operators implemented in LittleDarwin listed in Table 3.1. Each of these mutation operators replaces a syntactic feature with another. While in many mutation tools, several mutants are created from the same statement, LittleDarwin creates only one mutant per statement. For example, $a + b$ can be changed into $a - b$, $a * b$, or a/b , but in LittleDarwin $a + b$ is only replaced by $a - b$. These operators are based on the reduced-set of mutation operators that were demonstrated by Offutt et al. to be capable of creating similar-strength test suites as the full set of mutation operators [49]. Since the number of mutation operators of LittleDarwin is limited, it is possible that no mutants are generated for a class that lacks mutable statements. In practice, we observed that usually only very small compilation units (e.g. interfaces, and abstract classes) are subject to this condition.

In addition to these mutation operators, there are four experimental mutation operators in LittleDarwin that are designed to simulate null type faults. These mutation operators along with the faults they simulate are provided in Table 3.2. We included these mutation operators based on the conclusions offered by Osman et al. [92]. In their study, they discover that the null object is a major source of software faults. The null type mu-

Table 3.1: LittleDarwin Mutation Operators

Operator	Description	Example	
		Before	After
AOR-B	Replaces a binary arithmetic operator	$a + b$	$a - b$
AOR-S	Replaces a shortcut arithmetic operator	$++a$	$--a$
AOR-U	Replaces a unary arithmetic operator	$-a$	$+a$
LOR	Replaces a logical operator	$a \& b$	$a b$
SOR	Replaces a shift operator	$a >> b$	$a << b$
ROR	Replaces a relational operator	$a >= b$	$a < b$
COR	Replaces a binary conditional operator	$a \&\& b$	$a b$
COD	Removes a unary conditional operator	$!a$	a
SAOR	Replaces a shortcut assignment operator	$a * = b$	$a / = b$

Table 3.2: Null Type Faults and Their Corresponding Mutation Operators

Fault	Mutation Operator	Description
Null is returned by a method	NullifyReturnValue	If a method returns an object, it is replaced by <code>null</code>
Null is provided as input to a method	NullifyInputVariable	If a method receives an object reference, it is replaced by <code>null</code>
Null is used to initialize a variable	NullifyObjectInitialization	Wherever there is a new statement, it is replaced with <code>null</code>
A null check is missing	RemoveNullCheck	Any binary relational statement containing <code>null</code> at one side is negated

tation operators are able to simulate such faults, and consequently assess the quality of the test suite with respect to them. These mutation operators cover fault-prone aspects of a method: *NullifyInputVariable* mutates the method input, *NullifyReturnValue* mutates the method output, and *NullifyObjectInitialization* and *RemoveNullCheck* mutate the statements in method body.

3.3.4 Design Characteristics

To foster mutation testing in industrial setting it is important to have a tool able to work on large and complex systems. Moreover, to allow researchers to use real-life projects as the subjects of their studies, it is also important to provide a framework that is easy to extend. In this section, we show to what extent LittleDarwin, and its main alternatives, can satisfy these requirements. As alternatives, we use PITest [75], Javalanche [73], and MuJava [1], since they are popular tools used in literature. In Table 3.3, we summarize the design highlights.

Compatibility with Major Build Systems. To make the initial setup of a mutation testing tool easier, it needs to work with popular build systems for Java programs. LittleDarwin executes the build system rather than integrate into it, and therefore, can readily support various build systems. In fact, the only restrictions imposed by LittleDarwin

Table 3.3: Comparison of Features in Mutation Testing Tools

Features		LittleDarwin	PITest [93]	Javalanche [73]	MuJava [94]
Compatibility with	Maven	✓	✓	×	×
	Ant	✓	✓	×	×
	Gradle	✓	✓	×	×
	Others	✓	×	×	×
Support for Complex Test Structures		✓	×	×	×
Optimized for Performance		×	✓	✓	✓
Optimized for Experimentation		✓	×	×	×
Tested on Large Systems		✓	✓	✓	×
Ability to Retain Detailed Results		✓	×	×	✓
Open Source		✓	✓	✓	✓

are: (i) the build system must be able to run the test suite, and (ii) the build system must return non-zero if any tests fail, and zero if it succeeds. PITest address the challenge via integration into the popular build systems by means of plugins. At the time of writing it supports Maven³, Ant⁴, and Gradle⁵. Javalanche and MuJava do not integrate in the build system.

Support for Complex Test Structures. One of the difficulties of performing mutation testing on complex Java systems is to find and execute the test suite correctly. The great variety of testing strategies and unit test designs generally causes problems in executing the test suite correctly. LittleDarwin overcomes this problem thanks to a loose coupling with the test infrastructure, instead relying on the build system to execute the test suite. Other mutation testing tools reported in Table 3.3 have problems in this regard.

Optimized for Performance. LittleDarwin mutates the source code and performs the execution of the test suite using the build system. This introduces a performance overhead for the analysis. For each mutant injected, LittleDarwin demands a rebuild and test cycle on the build system. The rest of the mutation tools use byte code mutation, which leads to better performance.

Optimized for Experimentation. LittleDarwin is written in Python to allow fast prototyping [95]. To parse the Java language, LittleDarwin uses an Antlr4 parser. This allows us to rapidly adapt to the syntactical changes in newer versions of Java (such as Java 8). This parser produces a complete abstract syntax tree that makes the implementation of experimental features easier. In addition, the modular and multi-phase design of the tool allows reuse of each module independently. Therefore, it becomes easier to customize the tool according to the requirements of a new experiment. The other mutation tools work on byte code, and therefore do not offer such facilities.

³<https://maven.apache.org/>

⁴<https://ant.apache.org/>

⁵<https://gradle.org/>

Table 3.4: Comparison of Experimental Features in Mutation Testing Tools

Experimental Features	LittleDarwin	PITest	Javalanche	MuJava
Higher-Order Mutation	✓	×	×	×
Mutant Sampling	✓	×	×	✓
Subsumption Analysis	✓	×	×	×
Manual Mutation	✓	×	×	×
Incremental Analysis	✓	✓	×	×

Tested on Large Systems. LittleDarwin has been used in the past on software systems with more than 82 KLOC [33, 34]. PITest and Javalanche have been used in experiments with softwares of comparable size [73, 77]. We did not find evidence that MuJava has been tested on large systems.

Ability to Retain Detailed Results. PITest and Javalanche only output a report on the killed and survived mutants. However, in many cases this is not enough. For example, subsumption analysis requires the name of all the tests that kill a certain mutant. To address this problem, LittleDarwin retains all the output provided by the build system for each mutant, and allows for post-processing of the results. This also allows the researchers to manually verify the correctness of the results. MuJava provides an analysis framework as well, allowing for further experimentation [1].

Open Source. LittleDarwin is a free and open source software system. The code of LittleDarwin and its components are provided⁶ for public use under the terms of GNU General Public License version 2. PITest and MuJava are released under Apache License version 2. Javalanche is released into public domain without an accompanying license.

3.3.5 Experimental Features

In order to facilitate the means for research in mutation testing, LittleDarwin supports several features up to date with the state of the art. A summary of these features and their availability in the alternative tools is provided in Table 3.4. An explanation of each feature follows.

Higher-order Mutation. This feature is designed to combine two first-order mutants into a higher-order mutant. It is possible to link the higher-order mutants to their first-order counterparts after acquiring the results.

Mutant Sampling. This feature is designed to use the results for sampling experiments. LittleDarwin by default implements two sampling strategies: uniform, and weighted. The uniform approach selects the mutants randomly with the same chance of selection

⁶<https://github.com/aliparsai/LittleDarwin>

for all mutants. In the weighted approach, a weight is assigned to each mutant that is proportional to the size of the class containing the mutant. The given infrastructure also allows for the development of other techniques.

Subsumption Analysis. This feature is designed to determine the subsumption relationship between mutants. For each mutant, this feature can determine whether the mutant is subsuming or not, which tests kill the mutant, which mutants are subsuming the mutant, and which mutants are subsumed by the mutant. It is also capable of exporting the mutant subsumption graph proposed by Kurtz et al. for each project [69, 96].

Manual Mutation. This feature allows the researcher to use their manually created mutants with LittleDarwin. LittleDarwin is capable of automatically matching the mutants with the corresponding source files, and creating the required structure to perform the analysis. For example, this is useful in case the mutants are created with a separate tool.

Incremental Analysis. This feature allows the tool to be used in a continuous integration environment without the need of re-executing the analysis on all mutants after each integration step. Using this feature, developers can limit the analysis to the classes that have either recently changed or are related to the changes.

3.4 EXPERIMENTS

In this section, we provide a brief summary of the experiments we already performed using the experimental features of LittleDarwin on large and complex systems.

Mutation Testing of a Large and Complex Software System. We used LittleDarwin to analyze a large and complex safety critical system for Agfa HealthCare. Our attempts to use other mutation testing tools failed due to the complex testing structure of the target system. Due to this complexity, these tools were not able to detect the test suite. This is because (i) the project used OSGI⁷ headers to dynamically load modules, and (ii) the test suite was located in a different component, and required several frameworks to work. The loose coupling of LittleDarwin with the testing structure allowed us to use the build system to execute the test suite, and thus, successfully perform mutation testing on the project. For more details on this experiment, including the specification of the target system, and the run time of the experiment, please refer to Parsai’s master’s thesis [89].

Experimenting Up to Date Techniques on Real-Life Projects. LittleDarwin was used to perform three separate studies using the up to date techniques reported in Table 3.4. We were able to perform these studies on real-life projects.

In our study on random mutant sampling, we noticed that related literature have

⁷<https://www.osgi.org/developer/specifications/>

two shortcomings [33]. They focus their analysis at project level and they are mainly based on toy projects with adequate test suites. Therefore, we evaluated random mutant sampling at class level, and on real-life projects with non-adequate test suites. We used LittleDarwin to study two sampling strategies: uniform, and weighted. We highlighted that the weighted approach increases the chance of inclusion of mutants from classes with a small set of mutants in the sampled set, and reduces the viable sampling rate from 65% to 47% on average. This analysis was performed on 12 real-life open source projects.

In our study on higher-order mutation testing, we used LittleDarwin to perform our experiments [34]. We proposed a model to estimate the first-order mutation coverage from higher-order mutation coverage. Based on this, we proposed a way to halve the computational cost of acquiring mutation coverage. In doing so, we achieved a strong correlation between the estimated and actual values. Since LittleDarwin retains the information necessary for post-processing the results, we were able to analyze the relationship between each higher-order mutant and its corresponding first-order mutants.

We performed a study on simulating the null type faults which is currently under peer-review. In this study, we show that mutation testing tools are not adequate to strengthen the test suite against null type faults in practice. This is mainly because the traditional mutation operators of current mutation testing tools do not model null type faults. We implemented four new mutation operators in LittleDarwin to model null type faults explicitly, and we show how these mutation operators can be operatively used to extend the test suite in order to prevent null type faults. Using LittleDarwin, we were able to analyze the test suites of 15 real-life open source projects, and describe the trade offs related to the adoption of these operators to strengthen the test suite. We also used the mutant subsumption feature of LittleDarwin to perform redundancy analysis on all 15 projects.

Pilot Experiment. We performed a pilot experiment on a real life project in order to compare LittleDarwin with two of its alternatives: PITest and Javalanche. In this experiment, we used Jaxen⁸ as the subject, since it has been used before to evaluate Javalanche by its authors [97]. Jaxen has 12,438 lines of production code, and 7,539 lines of test code. Table 3.5 shows the results of our pilot experiment. As we can see, even though LittleDarwin creates the least number of mutants, it is still slowest per-mutant. This is mainly because PITest and Javalanche both filter the mutants prior to analysis based on statement coverage. In addition, LittleDarwin relies on the build system to run the test suite, which introduces per-mutant overhead.

Sanity Check Experiment. In order to assess the trustworthiness of LittleDarwin, a small sanity check experiment was conducted where we compared the results of LittleDarwin with those of PITest and JaCoCo. Even though the mutation operators of PITest

⁸<http://jaxen.org/>

Table 3.5: Pilot Experiment Results

Tool	Generated Mutants	Killed Mutants	Mutation Coverage	Analysis Time	Per-mutant Time
LittleDarwin	1,390	805	57.9%	2h23m45s	6.21s
PITest	4,315	2,145	49.8%	1h13m13s	1.02s
Javalanche	9,285	4,442	47.8%	1h35m23s	0.62s

are different from those of LittleDarwin, the overall assessment of the test suite should match in principle. The system under investigation during the pilot study was AddThis Codec⁹, a serialisation library for Java. This project has 46 classes in total, which is small enough to allow for manual inspection of the results.

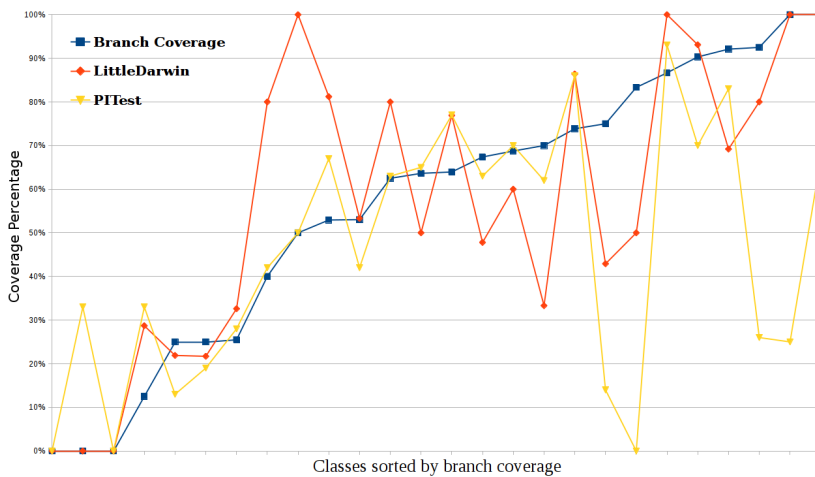


Figure 3.4: Comparing mutation coverage from PITest, and LittleDarwin, and branch coverage from JaCoCo

PITest generated a total number of 1,038 mutants of which 586 were killed by the test suite; resulting in a mutation coverage of 56%. LittleDarwin created 476 mutants with 295 of them killed; which lead to a mutation coverage of 62% for the whole project. Figure 3.4 shows the mutation coverage values calculated by PITest and LittleDarwin and the branch coverage values calculated by JaCoCo for each of the 46 classes. As one can see, for most of the classes, the mutation coverage are close to one another. Manual inspection confirmed that for those cases, PITest and LittleDarwin indeed create similar mutants. Also, manual inspection revealed that the classes with the highest difference in mutation coverage had only few mutants generated in total. For these classes, the small differences between the mutants created by each tool can result in a big impact on mutation coverage.

⁹<https://github.com/addthis/codec>

3.5 CONCLUSION

We presented LittleDarwin, a mutation testing framework for Java. On the one hand, it can cope with large and complex software systems. This lets LittleDarwin foster the adoption of mutation testing in industry. On the other hand, the tool is written in Python and released as an open source framework, namely it enables fast prototyping, and the addition of new experimental components. From this point of view, LittleDarwin shows its keen interest in representing an easy to extend framework for researchers on mutation testing. Combining these aspects allows researchers to use real-life projects as the subjects of their studies.

In the current version, LittleDarwin is compatible with major build systems, supports complex test structures, can work with large systems, and retains lots of useful information for further analysis of the results. Moreover, it already includes the following experimental features: higher-order mutation, mutant sampling, mutant subsumption analysis, and manual mutation. Using these features, we have already performed four studies on real-life projects that would otherwise not have been feasible.

Evaluating Random Mutant Selection at Class-Level in Projects with Non-Adequate Test Suites



Evaluating random mutant selection at class-level in projects with non-adequate test suites

Ali Parsai, Alessandro Murgia, and Serge Demeyer

In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE 2016)*, 11:1–11:10. Limerick, Ireland. June, 2016.

URL: <https://doi.org/10.1145/2915970.2915992>.

This chapter was originally published in the *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE 2016)*.

CONTEXT

This chapter targets the first of our three identified problems, the performance problem. In particular, it aims to evaluate random mutant selection on real-life software. Random mutant selection is an easy-to-implement way of reducing the performance overhead of mutation testing, and therefore, it can be easily added as a feature to the tools that are already publicly available. Here we assume that a random set of mutants are selected out of all the mutants generated for a software project, and that the distribution of equivalent mutants is uniform in the software under test.

We extended the threats to validity section here compared to the published version.

ABSTRACT

Mutation testing is a standard technique to evaluate the quality of a test suite. Due to its computationally intensive nature, many approaches have been proposed to make this technique feasible in real case scenarios. Among these approaches, uniform random mutant selection has been demonstrated to be simple and promising. However, works on this area analyze mutant samples at project level mainly on projects with adequate test suites. In this paper, we fill this lack of empirical validation by analyzing random mutant selection at class level on projects with non-adequate test suites. First, we show that uniform random mutant selection underachieves the expected results. Then, we propose a new approach named weighted random mutant selection which generates more representative mutant samples. Finally, we show that representative mutant samples are larger for projects with high test adequacy.

4.1 INTRODUCTION

The quality of a test suite is of interest to researchers and practitioners since the early days of software testing. One of the extensively studied approaches to quantify the quality of a test suite is mutation testing [38]. Mutation testing provides a well-studied approach to measure the quality of a test suite, and it is proven to simulate the faults realistically [24, 81]. This is due to the fact that the faults introduced by each mutant are modeled after the common mistakes developers often make [31].

Although the idea of mutation testing has been introduced since 1977 [38, 98], it has not found widespread use in real scenarios due to its computationally intensive nature. Therefore several approaches have been proposed in order to make this technique feasible in industrial settings [30]. Among these approaches, random mutant selection is one of the easiest to implement with promising results [64]. In this approach, instead of using all of the generated mutants, only a randomly selected subset of all mutants is selected to perform the mutation testing.

A common way to compare the random mutant selection approaches is to evaluate their level of *effectiveness*. The effectiveness of the sampled set is generally calculated in two steps. First, a certain number of test suites are created, each capable of killing all mutants in the sampled set. Then, the mutation coverage of each test suite is calculated, and the effectiveness of the sampled set is evaluated as the average of the mutation coverage of all these test suites [63]. Using this procedure, the random mutant selection has been demonstrated to be effective in literature [61, 63, 64, 65]. Here the mutant selection is performed using a uniform distribution, that is to say, all mutants had the same chance of being selected in the sampled set. Wong et al. reported that even using a subset of 10% of all generated mutants there is only a decrease of 16% of the performance achievable using full set of mutants [61, 64]. Likewise, Zhang et al. reported that a sample reduced

to half the size of the full set of mutants is equally effective [63]. However, these studies suffer from two shortcomings. First of all, effectiveness gives a biased picture of the sampled set as it favors large classes with many mutants. Secondly, they presume that the test suite one starts from has a good mutation coverage. We explain these shortcomings in the next two paragraphs.

The effectiveness measure gives a biased view, because it is computed at project level and provides no information on how the mutation coverage of individual classes is affected by using the sampled set. There might be classes where no mutants are selected at all, and yet the effectiveness of the sampled set would not be affected at project level. This is contrary to common practice, where test suite quality metrics are often calculated per class [99]. In that sense, studies based on the metric effectiveness lack of an analysis of the *representativeness* of the sampled set of mutants; namely, how using a sampled set of mutants influences the mutation coverage calculated for each class.

The assumption that projects have an adequate test suite (thus a test suite that has a 100% mutation coverage), is not realistic in many real projects. In that sense, studies based on this assumption need to be replicated in realistic scenarios to verify their validity. Only Zhang et al. [65] address the second threat by performing an experiment using non-adequate test suites as well as adequate test suites. However, they only considered the overall mutation coverage in their criteria and did not investigate the effects of random mutant selection on the mutation coverage at class level.

In this study, we attempt to fill the lack of empirical evaluation on random mutant selection on real projects by following the Goal-Question-Metric approach [100]. We set as *object* the process of random mutant selection in software projects with non-adequate test suites. Our *purpose* is to evaluate the representativeness of random mutant selection at class and project level. We evaluate the representativeness for software projects with different level of test adequacy. Then, we propose a new approach named weighted random mutant selection with the purpose of improving the representativeness of the sampled set of mutants at class level. We also investigate how the level of test adequacy varies along with acceptable sampling rate; namely the rate at which representativeness of the sampled set reaches a certain “acceptable” level (section 4.3.2). The *viewpoint* is that of software testers and testing researchers, both are interested in finding smaller yet representative sets of mutants able to work in real case scenarios. The *environment* of this study consists of 12 open-source projects. For this reason, we pursue the following research questions:

- **RQ1:** When does *uniform* random mutant selection achieve an acceptable degree of representativeness of the full set of mutants? We evaluate the representativeness of uniform random mutant selection (from now on, referred to as the uniform approach)

with various sampling rates on 12 open-source projects. Even though we are in agreement with previous research on this topic [61, 63, 64, 65] that using random mutant selection at low rates is effective in estimating project level mutation coverage, yet, the sampled set of mutants does not accurately represent the full set of mutants in estimating mutation coverage at class level.

- **RQ2:** To what extent can we reduce the acceptable sampling rate while keeping the same degree of representativeness?

We introduce a simple heuristic to improve the representativeness of randomly selected mutants. We call the new approach *weighted* random mutant selection (from now on, referred to as the weighted approach). This approach reduces the size of the sampled set at which we achieve acceptable mutant representativeness.

- **RQ3:** How does the level of test adequacy affect the acceptable sampling rate?

We investigate the effects of test adequacy on the acceptable sampling rate. We discover that the more adequate the test suite is, the higher the acceptable sampling rate becomes.

The rest of the article is structured as follows. In Section 4.2, background information regarding the study is provided. In Section 4.3, the details of the setup of the case study is discussed. In Section 4.4, the results are analyzed. In Section 4.5 we discuss the threats that affect the results. In Section 4.6, we report the state of literature on this topic. Finally, we present the conclusion in Section 4.7.

4.2 BACKGROUND

This section provides background information about what mutant sampling is, and the tool we use for our study.

4.2.1 Mutant Sampling

To make mutation testing practical, it is important to reduce the time it needs to run. One way to achieve this is to reduce the number of mutants. A simple approach to mutant reduction is to randomly select a set of mutants. This idea was first proposed by Acree [60] and Budd [39] in their PhD theses. To perform random mutant selection, we do not need any extra information regarding the context of the mutants. This makes easier the implementation of the mutation testing tools. Because of this, and the simplicity of random selection procedure, its performance overhead is negligible as well. The random mutant selection can be performed uniformly, meaning that each mutant has the same chance of being selected. Otherwise, the random mutant selection can be enhanced by

Operator	Description	Example	
		Before	After
AOR-B	Replaces a binary arithmetic operator	$a + b$	$a - b$
AOR-S	Replaces a shortcut arithmetic operator	$++a$	$--a$
AOR-U	Replaces a unary arithmetic operator	$-a$	$+a$
LOR	Replaces a logical operator	$a \& b$	$a b$
SOR	Replaces a shift operator	$a >> b$	$a << b$
ROR	Replaces a relational operator	$a >= b$	$a < b$
COR	Replaces a binary conditional operator	$a \&\& b$	$a b$
COD	Removes a unary conditional operator	$!a$	a
SAOR	Replaces a shortcut assignment operator	$a * = b$	$a / = b$

Table 4.1: LittleDarwin mutation operators

using heuristics based on the source code.

The percentage of mutants that are selected determines the *sampling rate* for random mutant selection. Using a fixed sampling rate is common in literature [63, 64, 65]. It is also possible to determine the sampling rate dynamically while performing mutation testing. A method resembling the latter was proposed by Sahinoglu and Spafford to randomly select the mutants until the sample size becomes statistically appropriate [66]. They concluded that their model achieves better results due to its self-adjusting nature [31].

4.2.2 LittleDarwin

To perform our analysis, we used the LittleDarwin¹ mutation testing tool previously used by Parsai et al. [78, 89]. This tool can perform mutation testing in complex (or simple) environments. LittleDarwin creates mutants by manipulating source code, and keeps the information about generated mutants and the results of the analysis on a local database, allowing to analyze the results further by using subsets of the final result. This also allows for the manual filtering of *equivalent* mutants², namely, the mutants that keeps the semantics of the program unchanged, and thus cannot be killed by any test [40].

In its current version, LittleDarwin supports mutation testing of Java programs with in total 9 mutation operators. These mutation operators are an adaptation of the minimal set introduced by Offutt et al. [49]. The description of each mutation operator along with an example can be found in Table 4.1.

¹<http://littledarwin.parsai.net/>

²We did not filter the equivalent mutants (see Section 4.5).

erage for the sampled set, and the full set.

Previous studies investigating how to reduce the full set of mutants used the metric *effectiveness* [61, 101]. To calculate effectiveness, first a set of test suites (T) is created, that each member is a test suite capable of killing all non-equivalent mutants in the sampled set. The mutation coverage of each test suite is then calculated using the full set of mutants, and effectiveness is defined as the average of the mutation coverage for all test suites in T . This metric only takes into account the effectiveness of the sampled set at project level.

In this work, we propose a new metric called *representativeness* to evaluate the correlation of the mutation coverage for the sampled set, and the full set of the mutants. To compute the representativeness of the random sample, we use Pearson's ρ and Kendall's τ_b correlation coefficients. We first partition the set of mutants according to set of classes, then we calculate the mutation coverage for each class using the equation in Figure 4.1. The same procedure is performed using the sampled set of mutants. The results are then used to calculate correlation coefficients.

We analyze whether the values of mutation coverage are linearly correlated (using Pearson's ρ [102]), and whether the ranking order of results can be predicted by the mutation coverage calculated using sampled sets (using Kendall's τ_b [103, 104]). Each one of these correlation coefficients provide a different outlook on the results. Pearson's ρ evaluates the linear correlation between the mutation coverage values of each class. The higher the correlation between the two sets (sampled and full), the higher is the representativeness of the sampled mutants. Kendall's τ_b shows if the sampled set can accurately predict the ranking order of the classes based on mutation coverage. This is desired, for example, if prioritizing the classes based on mutation coverage is important for the user. This coefficient has been used previously by Zhang et al. [63, 65]. By using this criteria, we aim to compare two approaches from the viewpoint of a developer who intends to discover which class is in need of more testing.

An example of these criteria is shown in Figure 4.2. In this figure, the small rectangles represent the classes and the large rectangle represents the project. The percentages inside each rectangle shows the mutation coverage of that class. The mutation coverage calculated using sampled set is shown in red while the mutation coverage calculated using all mutants is shown in green. The correlation between these two sets of mutation coverage values are then calculated using ρ and τ_b .

$$\text{Mutation Coverage} = \frac{\text{Killed Mutants}}{\text{All Mutants}}$$

Figure 4.1: Mutation coverage equation

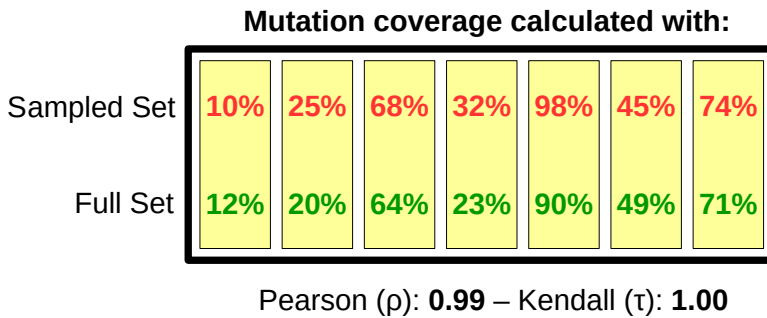


Figure 4.2: An example of calculation of correlation

Presenting mutation coverage as a percentage follows the same trend as other metrics such as branch coverage [45, 99]. We use mutation coverage as a metric to evaluate the quality of the test suite in the same manner. For object oriented programming languages like Java, test suite quality metrics are often calculated per unit, meaning that tools calculate the coverage per smaller units than the project as a whole [99]. Therefore, it is important that the results acquired from sampled set can emulate the results of full set of mutants per class. We consider a correlation value (ρ) of 0.75 the *critical point* after which two sets of mutation coverage values are strongly correlated. The choice of critical point is only to provide a reference, and slightly higher or lower values will not affect our conclusions (See Section 4.5). In our analysis, the size of these sets is always higher than 16 (granting a p-value lower than 0.01), with only the exception of JTerminal where the size is 6 (providing a p-value lower than 0.09). From now on, we define *acceptable* anything higher than the critical point, and as *acceptable sampling rate* the acceptable rate from where the degree of representativeness remains acceptable.

In order to evaluate the representativeness of the sampled set at project level,

we calculate the difference between the mutation coverage from the full set and the mutation coverage from the sampled set at various sampling rates. From now on, we refer to this difference as “distance”, namely the absolute value of the difference between mutation coverage of the full set and the mutation coverage of the sampled set. This gives us an idea on how close the sampled set can approximate the overall mutation coverage at various rates.

4.3.3 Algorithms

In order to calculate the representativeness of the sampled set of mutants, first we need to calculate the mutation coverage for each class using the sampled set. To do this, first we collect all the mutants belonging to the class from the sampled set. Then we use

the equation in Figure 4.1 to calculate the mutation coverage for that class. This procedure is repeated until each class has two mutation coverage measurements: one calculated using the full set, and another calculated using the sampled set. Then we calculate the correlation between these two sets of mutation coverage values using the correlation coefficients ρ and τ_b . This process is repeated 10 times in order to reduce the random noise. Finally, the average of all correlation values is reported for a specific sampling rate. By varying the sampling rate from 1% to 100%, we find the acceptable sampling rate for each project. The process of selection of mutants is performed in two different ways for our analysis. For the uniform approach, we use a random function to select N mutants from the full set. In the weighted approach, we first assign a weight proportional to inverse of the size of the class to each mutant, and then we use a roulette wheel algorithm to select the mutants. Assigning the inverse of the size avoids the overrepresentation of larger classes over smaller ones.

To do this, we first pick a random number r between 0 and the sum of all weights. Then, we add the weight of each mutant until this sum is greater than the random number r . The corresponding mutant is then selected. This procedure is repeated until the sample size reaches N . For the interested reader, the details of these algorithms are available online³.

4.4 RESULTS

In this section, we discuss the results of our study. For each research question, we first briefly describe our motivation, approach, and then our findings. In Table 4.3, for each project we report the acceptable sampling rate as determined by ρ and τ_b . We refer to Figures 4.7 and 4.8 to summarize the results of all research questions. In these figures, the horizontal axis is the sampling rate, and the vertical axis is the degree of representativeness. The red line shows the data for the uniform approach and the blue line shows the data for the weighted approach. We also draw a green line at the critical point to show where the sets of mutants start to be acceptable. While performing the study, we realized that the results from Pearson’s correlation and those of Kendall correlation were very close to each other. Therefore, it is unnecessary to report both in many parts of the analysis. So, wherever not specified, we refer to the Pearson’s correlation to report the degree of representativeness.

³<http://parsai.net/files/WRMSAFormalDef.pdf>

4.4.0 RQ1. *When Does Uniform Random Mutant Selection Achieve An Acceptable Degree of Representativeness of the Full Set of Mutants?*

Motivation. For different sampling rates, we want to evaluate the representativeness of the sampled set of mutants with respect to the full set of mutants. We want to perform this analysis at project and class level.

Approach. We perform an empirical study on 12 object oriented projects with different levels of test adequacy. For each project, we evaluate the degree of representativeness of the sampled set at project level. For this reason, we analyze the distance between the mutation coverage calculated from the sampled set, and the full set for sampling rates between 1% and 100%. Then, we calculate the degree of representativeness of the sampled set of mutants for different sampling rates at class level.

Findings. Zhang et al. [65] states that the uniform approach for adequate as well as non-adequate test suites provides near-perfect results with a low sampling rate at project level. Figure 4.3 shows the distance between the mutation coverage calculated from sampled set and full set. In this figure, we observe that the average distance over all of our projects is below 2% with a sampling rate as low as 5%. Our analysis confirms Zhang's observation, namely **at project level, the acceptable degree of representativeness is achieved at a very low sampling rate.** The different level of representativeness achieved at project and class levels can be explained by the dominance of classes with larger sets of mutants in the overall mutation coverage. **At class level, the acceptable degree of representativeness is achieved for sampling rates that span from 36% to 88%** (37% to 83% using Kendall's correlation). Table 4.3 shows that in JDepend, the appropriate acceptable sampling rate is 36%, since from this rate we obtain an acceptable representativeness. For JSQLParser the acceptable sampling rate is 88%. These two projects represent the maximum variability of the acceptable sampling rate. On average, the acceptable sampling rate for the uniform approach is 63%.

Focusing on Figures 4.7 and 4.8, we can notice that for many projects the sampling rate has an almost linear relationship with the degree of representativeness (e.g.; VRaptor, PITest, and Commons Lang). Whereas, in some other projects, this relationship is logarithmic-like, with the degree of representativeness that becomes acceptable sooner (e.g.; JGraphT, and JDepend). During our investigation, we found that correlation between the acceptable sampling rate and the number of classes with mutants is 0.44. Visually, this correlation can be seen in Figure 4.5. All these results show that **at class level, the acceptable sampling rate is project-dependent.** Moreover, we can see that difference between the number of mutants in the sampled set and the full set does not justify the degree of representativeness.

This result was unexpected since the analysis at project level reported that the uniform

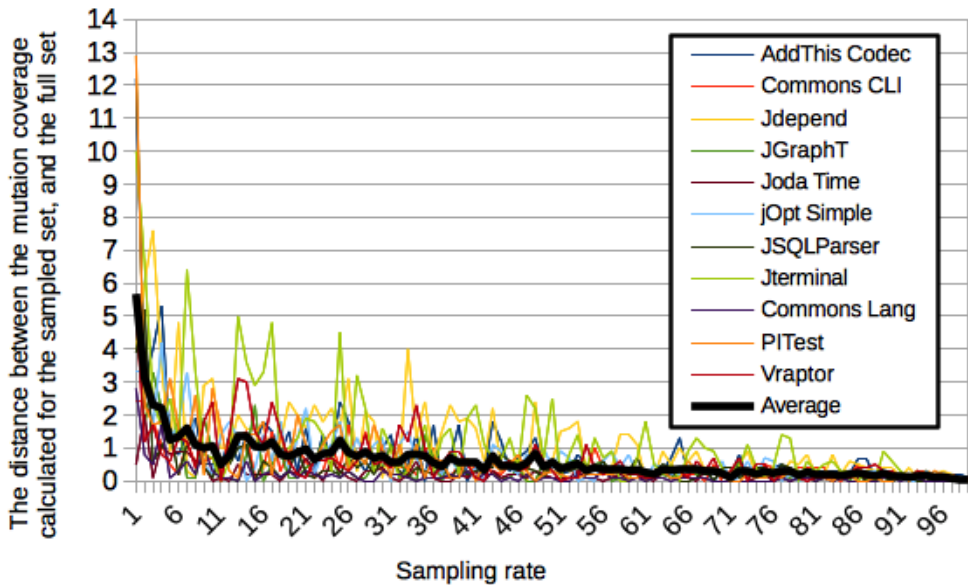


Figure 4.3: The distance between mutation coverage calculated from sampled set and full set for sampling rates from 1% to 100%

approach was viable with acceptable sampling rate as low as 5%. Whereas, our analysis at class level strongly differs having a acceptable sampling rate of 65% on average. From this point of view, the uniform approach underachieves the expected results.

4.4.0 RQ2. To What Extent Can We Reduce the Acceptable Sampling Rate While Keeping the Same Degree of Representativeness?

Motivation. During the analysis of RQ1, we noticed that some classes were not represented at all in the sampled sets of mutants. Classes with large number of mutants dominate the sampled set because their mutants have a higher chance of being selected. Consequently, the representativeness is negatively affected, since the sampled set may not include any mutants from classes with a small number of mutants. Therefore, we want a new heuristic able to increase the chances to select mutants from such classes.

Approach. We introduce a new heuristic that assigns more “weight” to the classes with a small number of mutants in order to increase the chance to select their mutants. Then we analyze to what extent our heuristic reduces the acceptable sampling rate with respect to the uniform approach (Table 4.3). Finally, we analyze if the acceptable sampling rate using our heuristic is project-dependent. For this reason, we analyze the relationship between sampling rate and the degree of representativeness for each project (Figures 4.7

and 4.8).

Findings. The uniform approach has a degree of representativeness that grows almost linearly with the sample size (Figures 4.7 and 4.8). This means that with a small sample size, it might not produce representative sampled sets. On the other hand, for a large sample the reduction in size would be negligible. **Keeping the same degree of representativeness, the average acceptable sampling rate for the weighted approach is 45%; which is 18% less than the uniform approach.** Almost in every project, the weighted approach goes close to the perfect representativeness around 75% sampling rate. Fixing the sampling rate, the weighted approach generates sampled sets with higher degree of representativeness compared to the uniform approach (Figures 4.7 and 4.8).

The reduction in sample size does not follow the same pattern in all projects. For example, in Apache Commons Codec (second row in the middle in Figures 4.7 and 4.8), using the uniform approach with a sampling rate higher than 72%, the representativeness remains acceptable (for both ρ and τ_b). For the weighted approach, the representativeness is already acceptable using only a sampling rate of 40% (a reduction of 32% of the sample size). On the other hand, JGraphT (top left in Figures 4.7 and 4.8) only shows a 5% reduction. By investigating this issue further, we discovered that for the weighted approach, the correlation between the acceptable sampling rate and the number of classes with mutants is 0.72 with a p-value of 0.008 (Figure 4.5), which is higher than the one achieved in the uniform approach. This happens because the more classes there are, the larger the sample needs to be in order to include mutants from all classes. We also find that the reduction of the sample size has a correlation of 0.57 (p-value 0.053) and 0.64 (p-value 0.025) respectively with the standard deviation (σ) and the average of the sizes (μ) of mutant sets for each class (Figure 4.4). This happens because if the size of classes are close to each other, the weights would be close as well. As a consequence, the weighted approach would behave like the uniform approach. Comparing these factors for JGraphT and Apache Commons Codec, we discover that for the former σ and μ are equal to 16.2, and 12.4 respectively, while for the latter σ and μ are equal to 83.3, and 48.2 respectively. These values are in agreement with our analysis.

4.4.0 RQ3. *How Does the Level of Test Adequacy Affect the Acceptable Sampling Rate?*

Motivation. During the investigation of RQ1 and RQ2, we noticed that the acceptable sampling rate is project-dependent. In literature [61, 63, 64], the analysis of sampling rates are done mostly at project level considering projects with adequate test suites. For this reason, we want to analyze the influence of the level of test adequacy on acceptable sampling rate for each project using class level criteria.

Approach. To evaluate the level of test adequacy we rely on mutation coverage as a

Project	Uniform		Weighted	
	$\rho > 0.75$	$\tau_b > 0.75$	$\rho > 0.75$	$\tau_b > 0.75$
Commons CLI	63%	69%	31%	39%
JSQParser	88%	83%	81%	77%
jOpt Simple	56%	65%	33%	38%
Commons Lang	79%	63%	50%	44%
Joda Time	58%	64%	45%	48%
Commons Codec	72%	72%	39%	40%
VRaptor	78%	79%	68%	68%
JGraphT	42%	52%	37%	48%
AddThis Codec	57%	67%	41%	50%
PITest	73%	74%	64%	67%
JTerminal	52%	53%	26%	33%
JDepend	36%	37%	21%	28%
Average	63%	65%	45%	47%

Table 4.3: acceptable sampling rate for uniform and weighted approaches

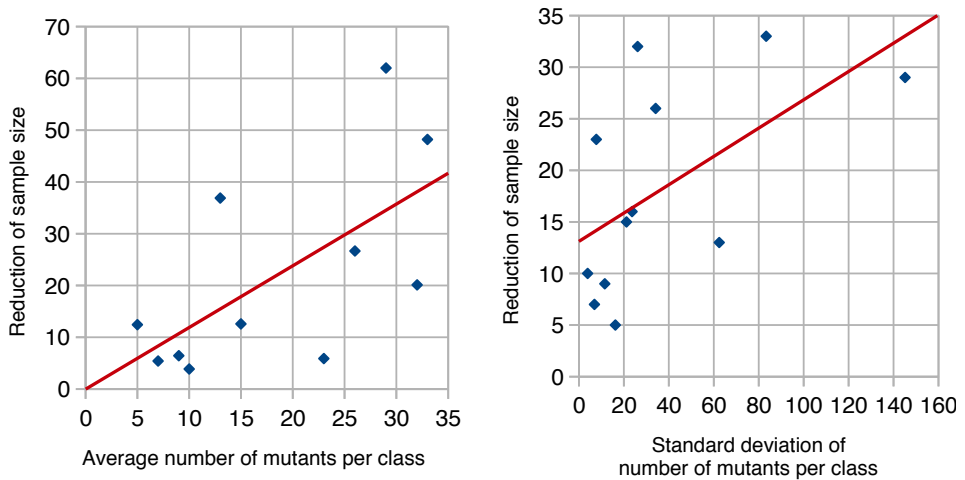


Figure 4.4: How the number of mutants per class affect the reduction of the sample size

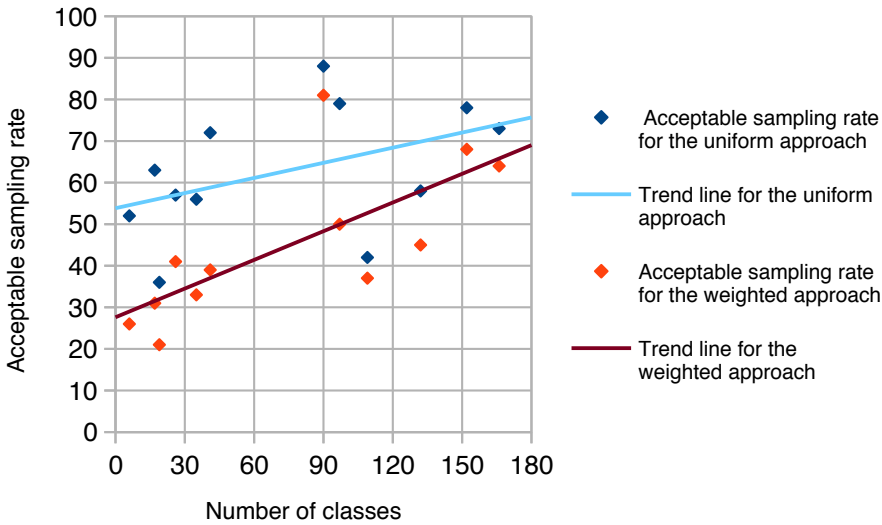


Figure 4.5: How the number of classes with mutants affects the acceptable sampling rate

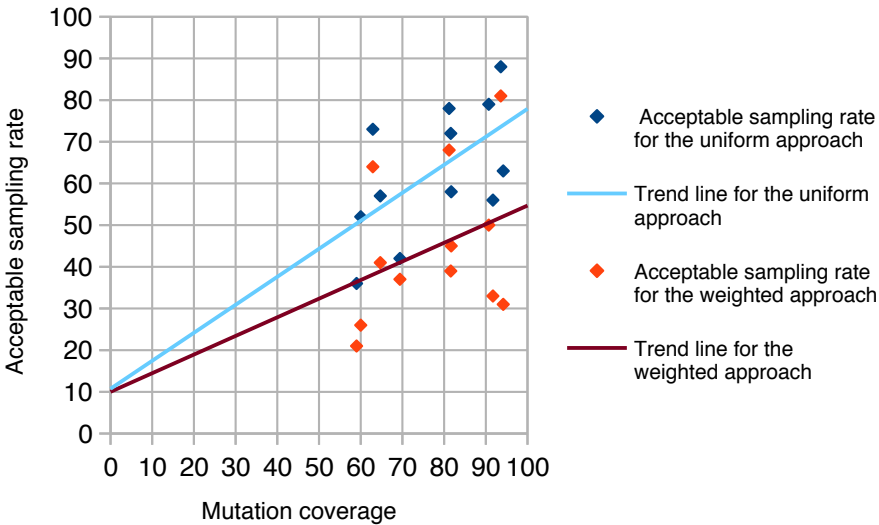


Figure 4.6: How mutation coverage affect the acceptable sampling rate

metric (Table 4.2). Test suites are considered as non-adequate if their mutation coverage is lower than 100%. In Table 4.3, we sort projects according to the level of test adequacy to check if and to what extent it influences the acceptable sampling rate. In Figure 4.6, we plot how test adequacy affects the acceptable sampling rate.

Findings. Figure 4.6 shows that **for the uniform approach and (to a lesser extent) for the weighted approach, the acceptable sampling rate increases with the test adequacy.** This result means that achieving an acceptable degree of representativeness

requires higher acceptable sampling rate in projects with higher level of test adequacy. This can be explained considering classes without mutants in the sampled set. If these classes had a high level of test adequacy, then their absence in sampled set would have a high negative impact on the degree of representativeness. Therefore, a larger sample is needed to be sure that all the classes are represented. For example, as seen in Table 4.3, JSQLParser and Apache Commons Lang have a acceptable sampling rate of 88% and 79% respectively, even though the level of test adequacy is higher than 90% in both projects. This effect is mitigated by using the weighted approach. For this reason, this behavior is less evident than in the uniform approach.

4.5 THREATS TO VALIDITY

To describe the threats to validity we refer to the guidelines reported by Yin [105].

Threats to **internal validity** focus on confounding factors that can influence the obtained results. These threats stem from potential bugs hidden inside the algorithms used for sampling mutants or in LittleDarwin. We consider this chance—even if possible—limited. The pseudo-code for random mutant selection is explained in our paper and its implementation has been carefully reviewed by the first author. The code of LittleDarwin has been already checked and tested in several case studies [78, 89]. Finally, the code of LittleDarwin along with all the raw data of the study is publicly available for download in the replication package.⁴

Threats to **external validity** correspond to the generalizability of our results. In our analysis we use only 12 open source projects. We mitigate this threat by using projects which differ for number of contributors, size and adequacy of the test suite. Yet, it is desirable to replicate this study using more projects, especially the ones belonging to industrial settings. A second threat stems from the limited set of mutation operators used in mutation testing process. Since mutation coverage depends on the mutants used, the sampling process and its accuracy is bounded to mutation operators that generate them. The impact of this threat is limited since we use the standard set of mutation operators which is representative of mistakes commonly introduced by developers and typically supported by many mutation testing tools [78]. In addition, since both uniform and weighted random selection was performed using the same set of mutation operators, the impact of extra mutation operators on the results of the study is minimal.

Threats to **construct validity** are concerned with how accurately the observations describe the phenomena of interest. In our case, this depends on the set of metrics adopted to evaluate the algorithms for random mutant selection. To measure to what extent the sampled mutants are representative of all possible mutants, we use the correlation be-

⁴<http://parsai.net/files/research/ReplicationPackage.7z>

CHAPTER 4. EVALUATING RANDOM MUTANT SELECTION AT CLASS-LEVEL IN PROJECTS WITH NON-ADEQUATE TEST SUITES

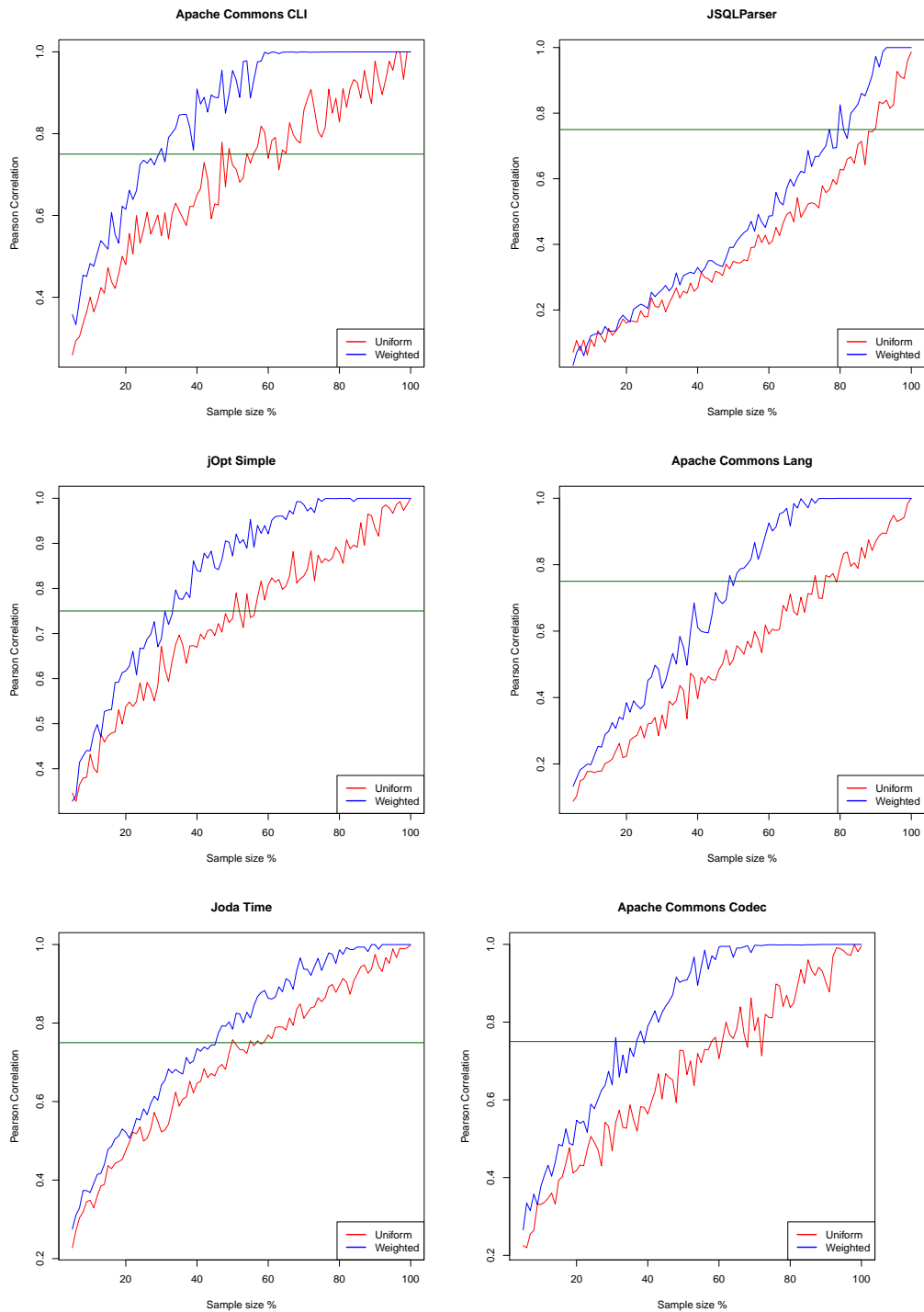


Figure 4.7: Pearson correlation between sampled and all mutants. The data reported in these figures is discrete. However for better visualization, the points are connected together.

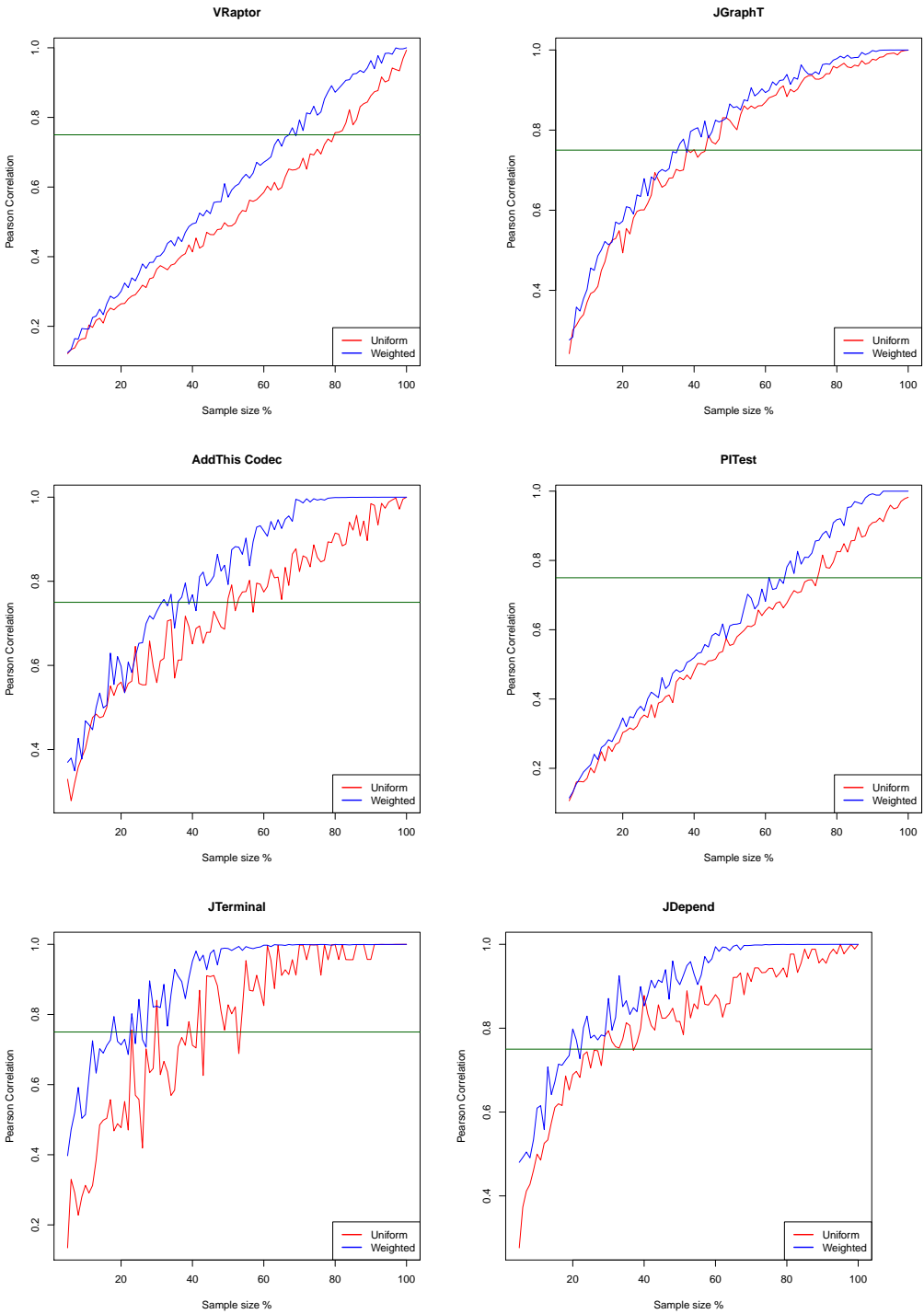


Figure 4.7: (Continued) Pearson correlation between sampled and all mutants. The data reported in these figures is discrete. However for better visualization, the points are connected together.

CHAPTER 4. EVALUATING RANDOM MUTANT SELECTION AT CLASS-LEVEL IN PROJECTS WITH NON-ADEQUATE TEST SUITES

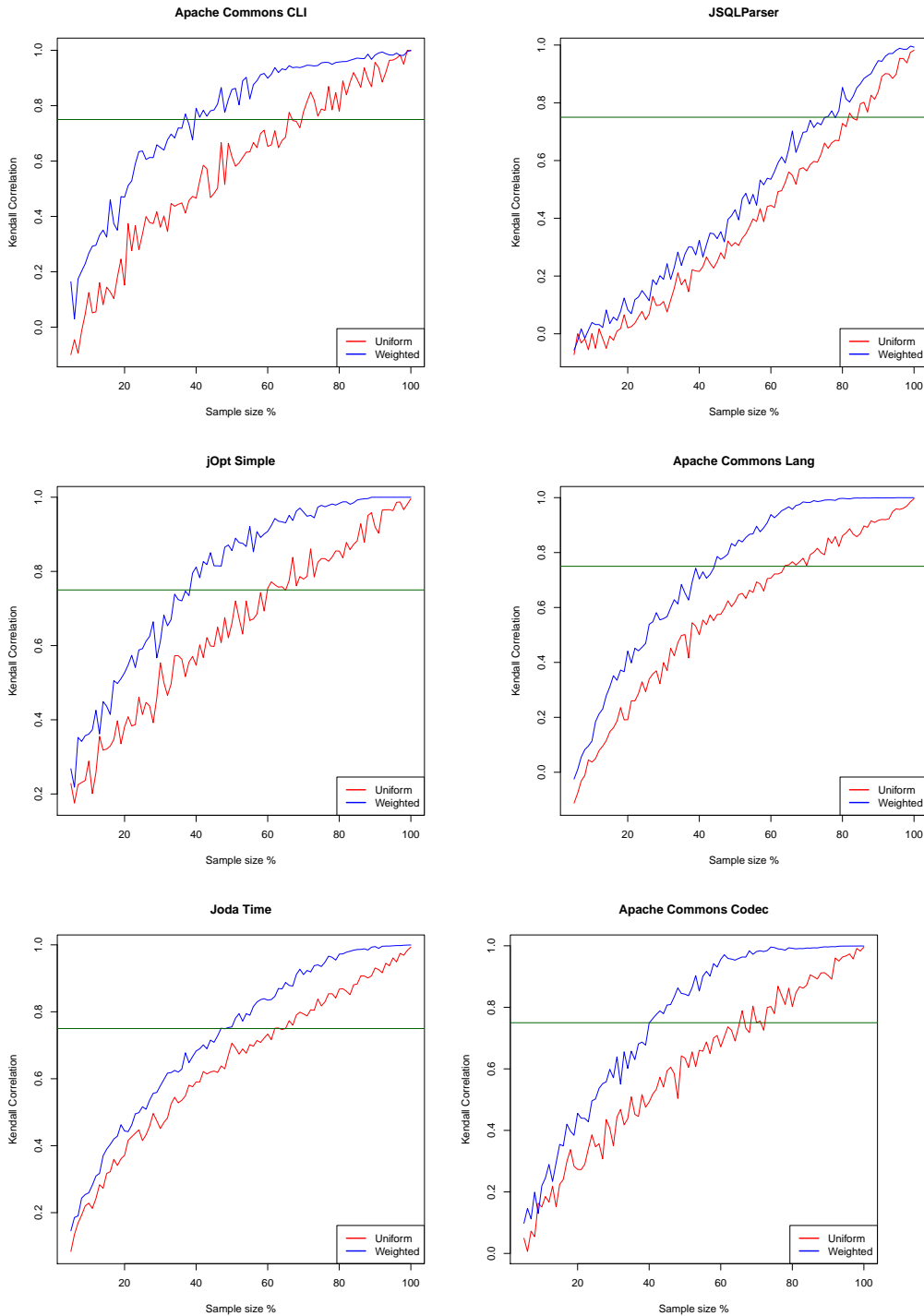


Figure 4.8: Kendall correlation between sampled and all mutants. The data reported in these figures is discrete. However for better visualization, the points are connected together.

4.5. THREATS TO VALIDITY

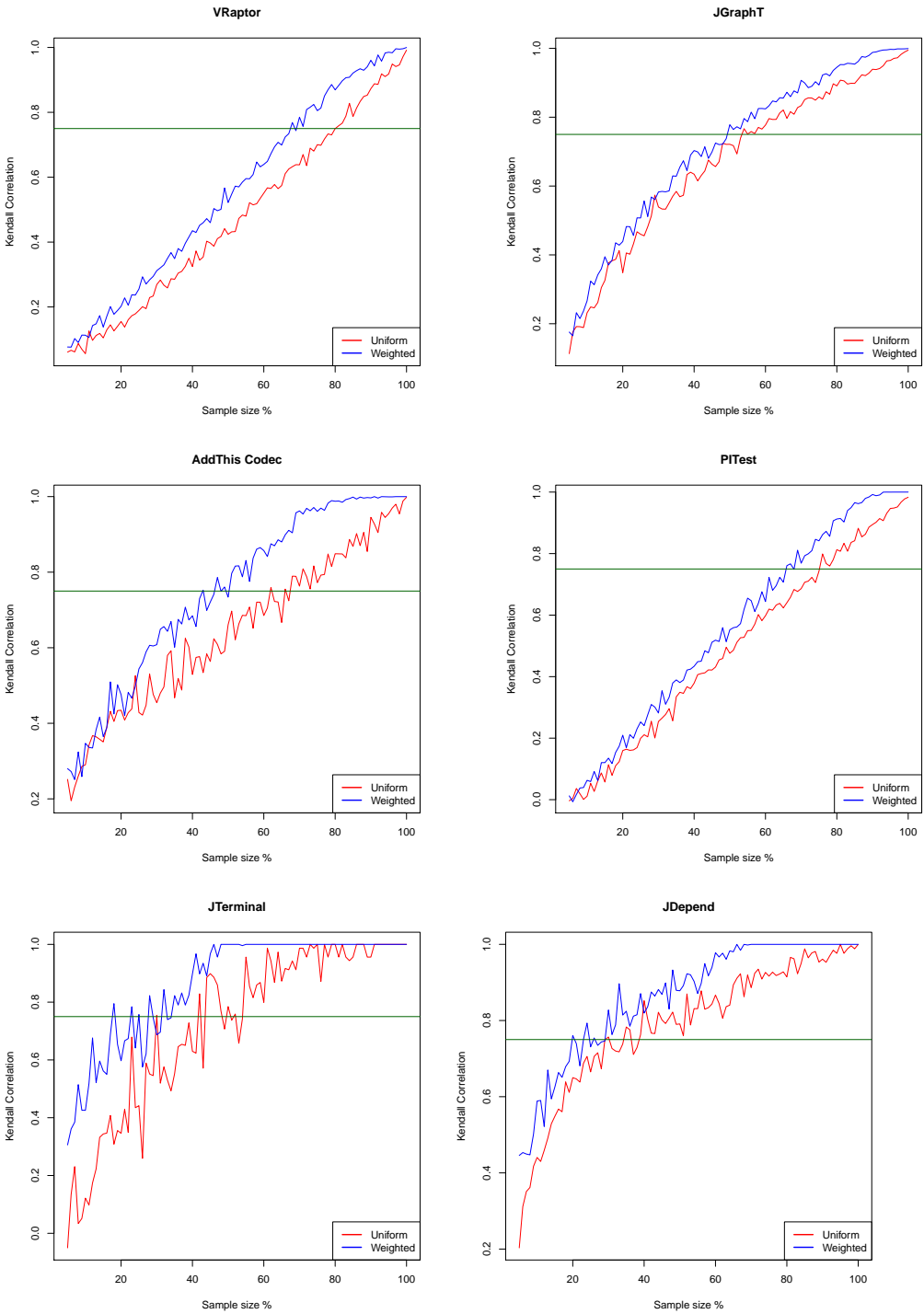


Figure 4.8: (Continued) Kendall correlation between sampled and all mutants. The data reported in these figures is discrete. However for better visualization, the points are connected together.

tween mutation coverage values calculated using sampled set and full set of mutants, thus emulating the way coverage metrics are commonly used in industry. We calculate correlations using well known metrics that have been used in literature numerous times. Even though we define the critical point after which the correlation is strong at 0.75, our results still hold if we choose slightly different thresholds. Due to often large sample sizes, the p-value for correlations were often much smaller than 0.001, and therefore we decided not to mention them explicitly. Sampling at project-level is another threat to construct validity. However, the alternative of sampling at class-level was not viable, since it is not possible to guaranty a certain size for the sampled set at project level.

As for filtering the equivalent mutants, we chose not to categorically remove a group of mutants, because the distribution of equivalent mutants in the sampled sets theoretically remains the same as the distribution in all mutants. Since, equivalent mutants act as false positives [40], we can argue that our sampled sets contain roughly the same percentage of false positives as in all mutants. Because classifying a single mutant for a class that has only few mutants as a false negative is more costly than adopting few false positives, we prefer the latter based on common practice [91].

Threats to **conclusion validity** are concerned with the degree to which conclusions we reach about relationships in our data are reasonable. Since the provided rationals in Section 4.4 justify our conclusions, we assume there are no threats to conclusion validity.

4.6 RELATED WORK

Reduction of the number of mutants has been investigated in literature to reduce the computational cost of mutation testing. Mutant selection is a simple approach towards this goal. There are two main branches to mutant selection: operator-based mutant selection and random mutant selection.

Operator-based mutant selection has been studied in detail in literature. Mathur [106] first proposed an approach based on selecting the *sufficient* set of mutant operators. Wong et al. [64] examined a selective set of mutation operators (2 out of 22) and concluded that the results are similar to those of all mutation operators. Offutt et al. [49, 101] demonstrated through empirical experiments that five mutation operators are sufficient to emulate the full set of mutation operators. Barbosa et al. uses random mutant selection as a control technique to determine the sufficient set of mutation operators for C [107]. Siami Namin et al. [108] used a subset of mutation operators for Proteum [109] that generated only 8% of all mutants. Gligoric et al. [74] extended this topic to concurrent code, and propose a set of 6 mutation operators as the sufficient set for the concurrent code. Random mutant selection has not been investigated as deeply as operator-based mutant selection in the literature. The first mention of random mutant selection is in the works of Acree

et al. [60, 110] and Budd [39], however, they did not perform any extensive empirical research on this subject. Wong and Mathur [64] then examined the uniform approach with incremental steps and concluded that a selection rate of 10% is only 16% less effective than all mutants. Zhang et al. have shown that the uniform approach using half the mutants is as effective as using all mutants [63]. They also proposed a two-round random mutant selection in which first a mutation operator is selected, and then a mutant generated by that operator is randomly selected. They demonstrated that this approach is more reliable due to less variance in the selected random sets in different runs. However, they used only projects with adequate test suites as the subjects of their experiment. Zhang et al. show that the use of random mutant selection along with operator-based mutant selection produces more accurate results [65]. They also show that the uniform approach provides accurate overall mutation coverage values for the test suite. In these studies [63, 64, 65] they only consider the correlation between several sets of test suites calculating the overall mutation coverage for the project. Our study differs from the previous ones in these respects:

- Instead of analyzing the representativeness of the sampled set at project level, we analyze it at class level. In this way, we avoid domination of larger classes over smaller ones in the analysis of the representativeness at project level.
- We study the largest set of projects with non-adequate test suites. Such projects have different characteristics such as size, number of contributors, and level of test adequacy.

4.7 CONCLUSION AND FUTURE WORK

Mutation testing is a widely studied method to determine the adequacy of a test suite. However, its adoption in real scenarios is hindered by its computationally intensive nature. Several approaches have been proposed to make it feasible in industrial settings and among them random mutant selection shows promising results. Studies of random mutant selection have two shortcomings, they focus their analysis at project level and they are mainly based on projects with adequate test suites. In this study we attempt to fill this the lack of empirical evaluation. We analyze uniform and weighted approaches in the context of random mutant selection. We compare these approaches at class level using as baseline 12 projects with various levels of test suite adequacy, code base sizes, and contributors.

We highlight that the uniform approach underachieves the expected results when analyzed at class level. We show that the uniform approach is only viable using a sampling rate around 65% (on average). Moreover, the degree of representativeness of the sampled sets grows linearly with the increase of the sampling rate. We also show that the

average acceptable sampling rate for the weighted approach is 18% less than the uniform approach while keeping the same degree of representativeness. The reduction in sample size between these two approaches is correlated with the average, and the standard deviation of the number of mutants per class. We discover that acceptable sampling rate is correlated with the number of classes with mutants in the uniform (and to a higher extent) to the weighted approach. We discover that the lack of representation of a class with a small set of mutants in the sampled set affects the projects with higher test adequacy in a stronger manner. By using the weighted approach this problem is reduced since it increases the chance of inclusion of mutants from classes with small set of mutants in the sampled set. We discovered that the number of mutants per class is a relevant factor to create a representative sampled set. For this reason, we invite fellow researchers to explore the influence of other factors such as type and position of the mutants for increasing the degree of representativeness.

A Model to Estimate First-Order Mutation Coverage from Higher-Order Mutation Coverage



A Model to Estimate First-Order Mutation Coverage from Higher-Order Mutation Coverage

Ali Parsai, Alessandro Murgia, and Serge Demeyer

In *2016 IEEE International Conference on Software Quality, Reliability, and Security (QRS 2016)*, 365–373. Vienna, Austria. August, 2016.

URL: <https://doi.org/10.1109/qrs.2016.48>.

This chapter was originally published in the *2016 IEEE International Conference on Software Quality, Reliability, and Security (QRS 2016)*.

CONTEXT

This chapter targets the first of our three identified problems, the performance problem. In particular, it aims to ease the use of higher-order mutation testing in real-life scenarios by providing a method to estimate first-order mutation coverage. The use of higher-order mutation coverage allows for a faster analysis without significant loss of information. Here we make several simplifying assumptions about the distribution of mutants and the probability of their status. These assumptions are made in order to reach a simplified model good enough for estimation purposes.

We removed the notion of false positives and explicitly mention the undecidability of equivalent mutant detection problem here compared to the published version.

ABSTRACT

The test suite is essential for fault detection during software development. First-order mutation coverage is an accurate metric to quantify the quality of the test suite. However, it is computationally expensive. Hence, the adoption of this metric is limited. In this study, we address this issue by proposing a realistic model able to estimate first-order mutation coverage using only higher-order mutation coverage. Our study shows how the estimation evolves along with the order of mutation. We validate the model with an empirical study based on 17 open-source projects.

5.1 INTRODUCTION

The advent of agile processes with their emphasis on test-driven development [111] and continuous integration [80] implies that developers want (and need) to test their newly changed or modified classes or components early and often [6]. Therefore, the quality of the test suite is an important factor during the evolution of the software. One of the extensively studied methods to improve the quality of a test suite is mutation testing [38]. Mutation testing provides a repeatable and scientific approach to measure the quality of the test suite, and it is demonstrated to simulate the faults realistically [24, 81]. This is because the faults introduced by each mutant are modeled after the common mistakes developers often make [31]. Although the idea of mutation testing has been introduced in the late 1970s [38, 98], it has not found widespread use in industry due to its computationally expensive nature. Therefore, several approaches have been proposed in order to make this technique feasible in industrial settings [30]. Exploiting higher-order mutants instead of first-order mutants is one way used in literature to approach this problem [31]. Higher-order mutants can be created by partitioning the set of first-order mutants randomly, and combining first-order mutants in each partition into higher-order mutants. The benefits of higher-order mutation are twofold. First, higher-order mutants are less likely to generate equivalent mutants than first-order mutants [68, 112]. The detection of equivalent mutants is an undecidable problem due to Turing’s halting problem [113]. However, when an equivalent mutant is combined with a non-equivalent mutant, the resulting higher-order mutant is non-equivalent as well [84]. This means that second-order mutants are less likely to suffer from the equivalency problem [67]. Second, by using higher-order mutants, fewer mutants need to be evaluated [67]. For instance, combining first-order mutants two-by-two into second-order mutants would reduce the number of mutants to 50%. As a consequence, it saves close to half of the computational time and make more feasible the integration of mutation testing in continuous integration systems.

Despite the benefits, the adoption of higher-order mutants presents also drawbacks and limitations. When higher-order mutants are constructed in the aforementioned manner, the mutation coverage calculated using higher-order mutants is not as precise as the

one calculated using first-order mutants, with the former often overestimating the capabilities of a test suite. While using higher-order mutants allows us to evaluate less mutants, it also means that we lose information regarding the status of each underlying first-order mutant. This creates a limitation on the order of mutation, as the value of the lost information overcomes the value of the information at hand.

In this paper, we address these shortcomings. We propose a model able to estimate first-order mutation coverage requiring only the computation of higher-order mutation coverage and yet with a smaller chance of equivalency problem. We are interested in a *realistic* model, namely a model that explains in a reasonable manner how the first-order mutants coverage affects the higher-order mutant coverage¹. Initially, we evaluate the estimation of our model for the second-order mutants. Then, we extend the analysis to find out whether our model can be used with mutants of third or higher order. For each higher-order mutant, we also verify whether our model correctly describes their real behavior. More specifically, we look for side effects (e.g.; fault shifting and fault masking [68]) due to the combination of first-order mutants into higher-order mutants.

This work follows the Goal-Question-Metric paradigm [100]. The *goal* of this study is to build a model based on higher-order mutation coverage able to estimate the first-order mutation coverage. The behavior of the empirical data must be explained reasonably by such model. The *focus* is to explore the accuracy and the limitations of the model according to the order of mutation. The *viewpoint* is that of the team leaders and testing researchers, both interested in making mutation testing applicable in real case scenarios. The *environment* of this study consists of 17 open-source cases. For this reason, we pursue the following research questions:

- **RQ1.** To what extent the second-order mutants can be used in place of the first-order mutants for estimation purposes?
- **RQ2.** How does incrementing the order of mutation affect the accuracy of the estimation provided by the model?

The rest of the paper is structured as follows: Section 5.2 provides necessary background information about the subject of the study. Section 5.3 describes our proposed model. In Section 5.4 we explain the design of our case study. In Section 5.5 we present the results of our experiment. Section 5.6 contains the threats to the validity of our study, and our attempts to reduce them. Section 5.7 discusses the related studies, and their differences with ours. Finally, we conclude the paper in Section 5.8.

¹Several polynomial functions of order N can fit the empirical data. Yet, the underlying model may not be suitable to explain the behavior of the data.

5.2 BACKGROUND

In this section we present an overview of the background information needed to understand our proposed model. First we explain the mutation testing, then we discuss the equivalent mutants, and finally, we take a closer look at the higher-order mutants.

5.2.0 Mutation Testing

Mutation testing is the process of injecting faults into software, and counting the number of these faults that make at least one test fail. The idea of mutation testing was first mentioned in a class paper by Lipton and later developed by DeMillo, Lipton, and Sayward [38]. The first implementation of a mutation testing tool was done by Timothy Budd in 1980 [39].

The procedure for mutation testing is as follows: First, faulty versions of the software are created by introducing a single fault into the system (*Mutation*). This is done by applying a known transformation on a certain part of the code (*Mutation Operator*). After generating the faulty versions of the software (*Mutants*), the test suite is executed on each one of these mutants. If there is an error or failure during the execution of the test suite, the mutant is regarded as killed (*Killed Mutant*). On the other hand, if all tests pass, it means that the test suite could not catch the fault and the mutant has survived (*Survived Mutant*). The final result is calculated by dividing the number of killed mutants by the number of all non-equivalent mutants. This metric provides a detailed assessment of the quality of a test suite, as it makes sure that the kind of faults simulated by the mutation operators are covered by the tests and therefore reduces the chance of missing such faults in the final product. In addition of using mutation coverage, test developers can target surviving mutants. This allows killing mutants to be used as a test requirement.

5.2.0 Equivalent Mutants

If a mutant produces the same output as the original program for all input values, it is called an equivalent mutant. The creation of equivalent mutants is undesirable [40], but they are not easy to detect [41]. The creation of such mutants depends on the context of the program itself, for example, in Figure 5.1, the introduced change in *proc1* will change the outcome, while the same change in *proc2* does not. The error introduced in *proc2* is, therefore, not detectable by any test.

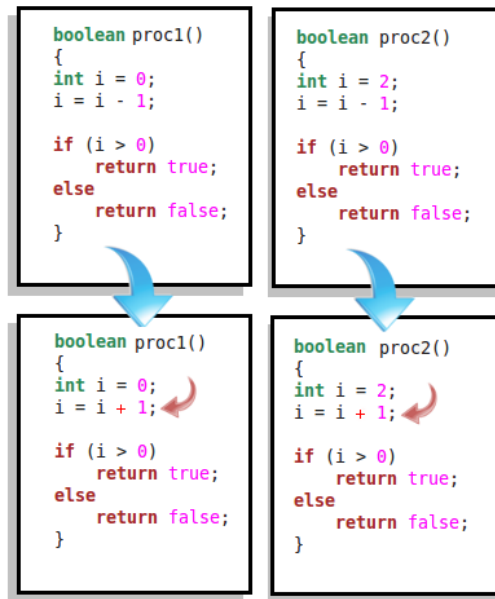


Figure 5.1: Example for emergence of equivalence due to the context

5.2.0 First-Order Mutants and Higher-Order Mutants

First-order mutants are the mutants generated by applying a mutation operator on the source code only once. By applying mutation operators more than once we obtain higher-order mutants. Higher-order mutants can also be described as a combination of several first-order mutants. Jia et al. [67] introduced the concept of higher-order mutation testing and discussed the relation between higher-order mutants and first-order mutants.

They divided the higher-order mutants into four categories based on the observed coupling effect [68]: *Expected*, *Worst*, *Fault Shift*, and *Fault Mask*. However, this categorization cannot be used to clarify the *unexpected status* of the higher-order mutants, mainly because it does not consider how the status of the higher-order mutant relates to the status of the underlying first-order mutants. Therefore, we propose the following categorization:

- **Expected.** The higher-order mutant is killed, and at least one of the underlying first-order mutants is killed; or the higher-order mutant has survived as well as all the underlying first-order mutants.
- **HK-FS.** The higher-order mutant is killed, even though all the underlying first-order mutants have survived.
- **HS-FK.** The higher-order mutant has survived, even though at least one of the under-

Table 5.1: LittleDarwin mutation operators

Operator	Description	Example	
		Before	After
AOR-B	Replaces a binary arithmetic operator	$a + b$	$a - b$
AOR-S	Replaces a shortcut arithmetic operator	$++a$	$--a$
AOR-U	Replaces a unary arithmetic operator	$-a$	$+a$
LOR	Replaces a logical operator	$a \& b$	$a b$
SOR	Replaces a shift operator	$a \gg b$	$a \ll b$
ROR	Replaces a relational operator	$a \geq b$	$a < b$
COR	Replaces a binary conditional operator	$a \&\& b$	$a b$
COD	Removes a unary conditional operator	$!a$	a
SAOR	Replaces a shortcut assignment operator	$a * = b$	$a / = b$

lying first-order mutants is killed.

When considering the overall percentage for each class, the mutants in categories HK-FS and HS-FK compensate the effects of each other. Therefore, if there are the same number of mutants in each of these categories, it is not visible in the final mutation coverage. The model must take into account these interactions among categories to correctly describe the real behavior of higher-order mutants.

5.2.0 LittleDarwin

To perform our analysis, we modified the LittleDarwin² mutation testing tool previously used by Parsai et al. [33, 78, 89] to generate higher-order mutants. LittleDarwin creates mutants by manipulating source code, and keeps the information about generated mutants and the results of the analysis in a local database, allowing to perform further analysis of the results.

In its current version, LittleDarwin supports mutation testing of Java programs with a total of 9 mutation operators. These mutation operators are an adaptation of the minimal set introduced by Offutt et al. [49]. The description of each mutation operator along with an example can be found in Table 5.1. LittleDarwin constructs higher-order mutants by combining two (or more) first-order mutants randomly at class-level. The information regarding the underlying first-order mutants are provided inside each higher-order mutant.

²<http://littledarwin.parsai.net/>

5.3 PROPOSED MODEL

In order to estimate the first-order mutation coverage from the higher-order mutation coverage (mutation coverage calculated using higher-order mutants) we follow these steps:

- We define a higher-order mutant as a combination of multiple first-order mutants:

$$h = m_1, m_2, \dots, m_n \quad (5.1)$$

- We assume that the higher-order mutant is killed if and only if at least one of the underlying first-order mutants are killed (Expected category).

The probability of a mutant ($P(\text{mutant})$) being killed is defined as the likelihood of choosing a random mutant with a killed status out of all mutants. The probability of a higher-order mutant being killed ($P(h)$) can be calculated from the probability of underlying first-order mutants ($P(m_i)$) in the following manner:

$$\begin{aligned} 1 - P(h) &= (1 - P(m_1)) \\ &\quad \times (1 - P(m_2|m_1)) \\ &\quad \dots \\ &\quad \times (1 - P(m_n|m_1, m_2, \dots, m_{n-1})) \end{aligned} \quad (5.2)$$

This simply means that the probability of a higher-order mutant being killed can be evaluated by an ordered evaluation of the probabilities of underlying first-order mutants being killed.

- After calculating the higher-order mutation coverage, we assume all higher-order mutants have the same probability of being killed equal to this mutation coverage³.
- For the sake of estimation, we assume all first-order mutants have the same equal probability of being killed, and this probability is equal to first-order mutation coverage. We assume these probabilities to be independent from each other.

$$P(m_1) = P(m_2|m_1) = \dots = P(m_n|m_1, m_2, \dots, m_{n-1})$$

To ease the calculation of mutation coverage, we simplify Equation 5.3 using the above assumptions:

³For example, imagine a bag of colored balls, in which 30% are red, and 70% are blue. Now if a ball is picked at random, there is a 30% probability that this ball is red.

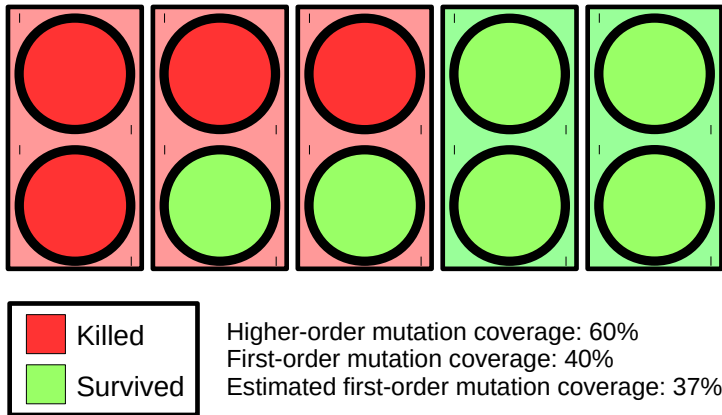


Figure 5.2: Example of the estimation provided by the model

$$1 - P(h) = (1 - P(m))^n \quad (5.3)$$

Using Equation 5.3, we assume $P(h)$ equals to the higher-order mutation coverage, i.e. any selected higher-order mutant from a set containing $n\%$ killed higher-order mutants is a killed one with the probability of $n\%$. From this equation, we derive the value for $P(m)$, which is the probability of a single first-order mutant being killed, and we use this value as our estimation for first-order mutation coverage (Equation 5.4).

$$P(m) = 1 - \sqrt[n]{1 - P(h)} \quad (5.4)$$

Figure 5.2 shows an example of this model; circles denote first-order mutants and rectangles are second-order mutants created by combining two first-order mutants. Out of 10 first-order mutants, 4 are killed, therefore the mutation coverage for this example is equal to 40%. However, 3 out of 5 higher-order mutants are killed, resulting in 60% coverage calculated using higher-order mutants. Considering our model, we estimate the first-order mutation coverage using Equation 5.4 as 37%.

5.4 CASE STUDY DESIGN

In order to evaluate our model, we performed first-, second-, third-, fourth-, fifth-, sixth-, and eighth-order mutation testing on 17 different Java projects. For these projects, 20849 first-order mutants were generated for a total of 1022 classes. In this section, we first describe our cases, and then we discuss how the study was performed.

5.4.0 Cases

We selected 17 open-source projects for our empirical study (Table 5.2). These projects are Joda Time⁴, Apache Commons Lang⁵, Apache Commons Codec⁶, AddThis Codec⁷, Apache Commons CLI⁸, Apache Commons FileUpload⁹, JSQParser¹⁰, JDepend¹¹, JGraphT¹², JTerminal¹³, VRaptor¹⁴, PITest¹⁵, HTTP Request¹⁶, jsoup¹⁷, Linq4J¹⁸, ScribeJava¹⁹, and jOpt Simple²⁰.

The selected projects differ in size of their production code, test code, number of commits, and team size to provide a wide range of possible scenarios. Moreover, they also differ in adequacy of the test suite. based on statement, branch, and mutation coverage (Table 5.2). All selected projects are written in Java, which is a widely used programming language in industry [7].

5.4.0 Model Evaluation

The accuracy of the model is affected by the accuracy of $P(m)$. This in turn depends on the number of mutants generated for the class in question (10 mutants would result in a probability with 0.1 accuracy, while 100 mutants improves this to 0.01). We define a threshold t as the number of minimum generated mutants for a class, and we filter out the classes with less than t generated mutants. The higher the threshold, the fewer the classes considered. This can be seen in Figure 5.3, where the x axis is the threshold, and the y axis is the number of remaining classes after applying the threshold. In our experiment, we decided to evaluate how the model behaves with different values of t in order to find a value that still keeps as many classes as possible, while filtering out less accurate data.

The first goal of our work is to find a model that estimates —as best as possible— the first-order mutation coverage using higher-order mutants coverage. The second one

⁴<http://www.joda.org/joda-time/>

⁵<http://commons.apache.org/proper/commons-lang/>

⁶<http://commons.apache.org/proper/commons-codec/>

⁷<http://github.com/addthis/codec>

⁸<http://commons.apache.org/proper/commons-cli/>

⁹<http://commons.apache.org/proper/commons-fileupload/>

¹⁰<https://github.com/JSQParser/JSQParser>

¹¹<http://www.clarkware.com/software/JDepend.html>

¹²<http://jgrapht.org/>

¹³<https://grahamedgecombe.com/projects/jterminal>

¹⁴<http://www.vraptor.org/>

¹⁵<http://pitest.org/>

¹⁶<http://kevinsawicki.github.io/http-request/>

¹⁷<http://jsoup.org/>

¹⁸<http://www.hydromatic.net/linq4j>

¹⁹<https://github.com/scribejava/scribejava>

²⁰<http://pholser.github.io/jopt-simple/>

Table 5.2: Projects sorted by mutation coverage

Project	Ver.	Size (LoC)		#C	TS	SC	BC	MC	#M
		Prod.	Test						
ScribeJava	1.3.0	2002	1530	525	69	43%	66%	94.9%	59
Apache Commons CLI	1.3.1	2665	3768	816	15	96%	93%	94.2%	342
JSQParser	0.9.4	7342	5909	576	19	81%	73%	93.6%	488
jOpt Simple	4.8	1982	6084	297	14	99%	97%	91.7%	206
Apache Commons Lang	3.4	24289	41758	4398	30	94%	90%	90.7%	6014
Joda Time	2.8.1	28479	54645	1909	42	90%	81%	81.7%	4870
Apache Commons Codec	1.10	6485	10782	1461	10	96%	92%	81.6%	1976
VRaptor	3.5.5	14111	15496	3417	65	87%	81%	81.2%	589
HTTP Request	6.0	1391	2721	446	15	94%	75%	78.4%	227
Apache Commons FileUpload	1.3.1	2408	1892	846	19	76%	74%	77.1%	354
jsoup	1.8.3	10295	4538	888	43	82%	72%	76.1%	1219
JGraphT	0.9.1	13822	8180	1150	31	79%	73%	69.4%	1356
AddThis Codec	3.3.0	3675	1342	249	4	69%	63%	64.7%	450
PITest	1.1.7	17244	19005	1044	19	79%	73%	62.9%	1070
JTerminal	1.0.1	687	250	8	2	66%	56%	60.0%	160
JDepend	2.9.1	2460	1053	18	2	59%	52%	59.0%	239
Linq4J	0.4	14307	3979	205	7	33%	66%	46.2%	1230

Acronyms:

Version (Ver.), Line of code (LoC), Production code (Prod.),
 Number of commits (#C), Team size (TS), Statement coverage (SC),
 Branch coverage (BC), Mutation coverage (MC), Number of Mutants (#M)

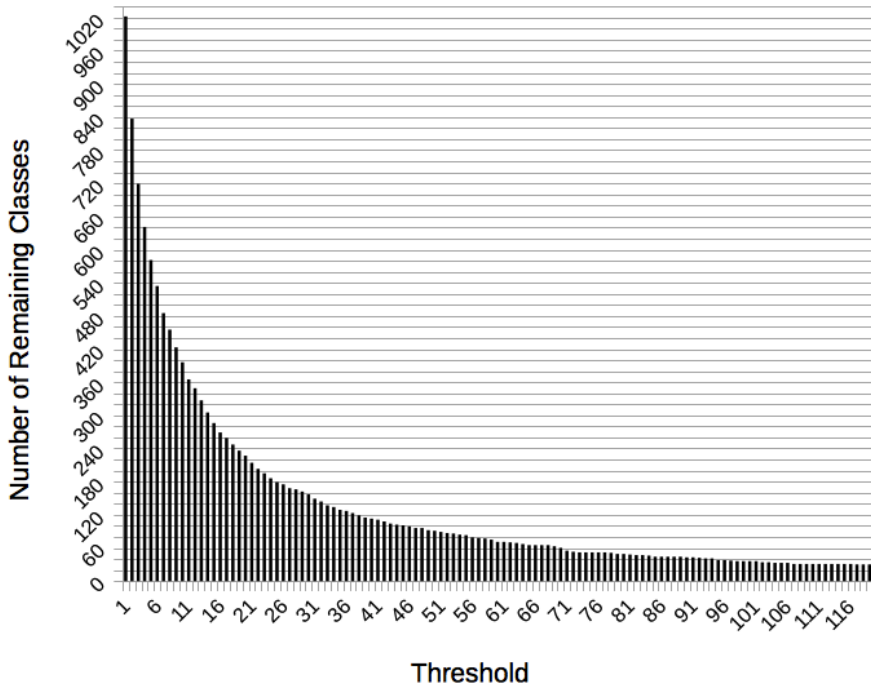


Figure 5.3: Number of remaining classes after applying threshold t

is to create a model able to justify the obtained estimation. For the first goal, a simple polynomial regression analysis is used in order to find the best fitting curve of the empirical data. The order of the polynomial function influences the quality of the fitting: the higher the order, the higher the chance that the curve (over)fits the all points. However, to satisfy the second goal, we also need to find a curve that describes the *real* behavior of the empirical data. For this reason, we cannot use a polynomial fitting curve of third or higher order. A polynomial curve of third order has 1 ($3 - 2$) inflections, namely there exist a range of values where an increment of the first-order coverage leads to a decrement of the second-order coverage. The latter event cannot happen since it would imply that the more first-order mutants are killed, the less higher-order mutant are killed. In our experiment, we compare the best second order fitting curve with the one provide by our model. This allows us to assess how far our model is from the optimal estimation.

5.5 RESULTS AND DISCUSSION

In this section, for each research question, we first discuss the motivation, then we explain our approach, and finally we present our findings.

5.5.0 RQ1. *To What Extent the Second-Order Mutants Can Be Used in Place of the First-Order Mutants For Estimation Purposes?*

Motivation. We are interested to verify whether the higher-order mutants can be used in place of the first-order mutants —with negligible drawbacks— for the estimation of the quality of the test suite. For the sake of clarity, this RQ analyzes the model based only on second-order mutants. If with this order the model performs poorly, then there would be no reason to extend the analysis to higher orders. We want to verify the applicability of the model, and for this reason, we inspect two aspects:

- *Estimation Accuracy.* The second-order mutation coverage overestimates the capabilities of the test suite, i.e. second-order mutants are easier to kill compared to first-order mutants. We need to empirically evaluate the accuracy of the estimations based on second-order mutation coverage.
- *Soundness of the Model.* Our model is based on the assumption that second-order mutants are killed *if and only if* at least one of the underlying first-order mutants is killed. We need to verify if this assumption holds with the empirical data.

For this reason, we break the first research question in two parts:

- a) *How accurately can our model estimate the first-order mutation coverage?*

Table 5.3: Parameters of the estimated curve

Parameter	Estimated Value	Standard Error	p-value
x^0	0.005342	0.010109	0.598
x^1	1.999221	0.037831	$< 2 \times 10^{-16}$
x^2	-1.008797	0.032452	$< 2 \times 10^{-16}$

Approach. We validate our model using 17 open-source projects. For all projects, we perform both first-order and second-order mutation testing. Then, we verify how the threshold t of the generated mutants affects the estimations. To evaluate the accuracy of the estimations, we calculate the coefficient of determination (R^2) between the results estimated by our model, and the empirical data [114, p. 78].

We compare the curve predicted by our model with the best fitting second-order curve obtained by using polynomial regression analysis on the empirical data. From this comparison, we can evaluate how close is the curve proposed by the model from the best possible one.

Findings. Figure 5.4 shows the value of R^2 between estimated results and empirical data for various values of the threshold t . The estimations achieve an R^2 value 0.85 with a threshold t as low as 10. Using such threshold, the model can still be applicable to 38.7% of the classes which account for 89.8% of the mutants. For higher thresholds, the R^2 value is always higher than 0.85. These results show that the model can provide a good accuracy in estimating the first-order mutation coverage from the second-order mutation coverage. Yet, it halves the computation time required.

Figure 5.5 shows the empirical data for $t = 10$ and the estimated curve. The blue dots denote a class with an x value of first-order mutation coverage and a y value of second-order mutation coverage. The red curve is the one predicted by our model: $y = -x^2 + 2x$. The curve determined by the regression analysis is $y = -1.008797x^2 + 1.999221x + 0.005342$. Table 5.3 shows the p-value for each estimated parameter. Considering the very low p-values calculated for each parameter, we can safely say that the trend of the empirical data is well represented by the best fitting curve. As we can see, the curve predicted by our model, and the one provided by the regression analysis are very close in terms of coefficients. This highlights that our model provides (almost) the best possible estimations. The accuracy of the estimation, however, changes according to the mutation coverage value. This is observed especially for classes with a very high second-order mutation coverage, in which there is less information available for the model. At its extreme, when all the higher-order mutants are killed, the model always estimates a 100% first-order coverage, even though some of the underlying first-order mutants might survive.

b) Is the modeled behavior of second-order mutants consistent with the empirical data?

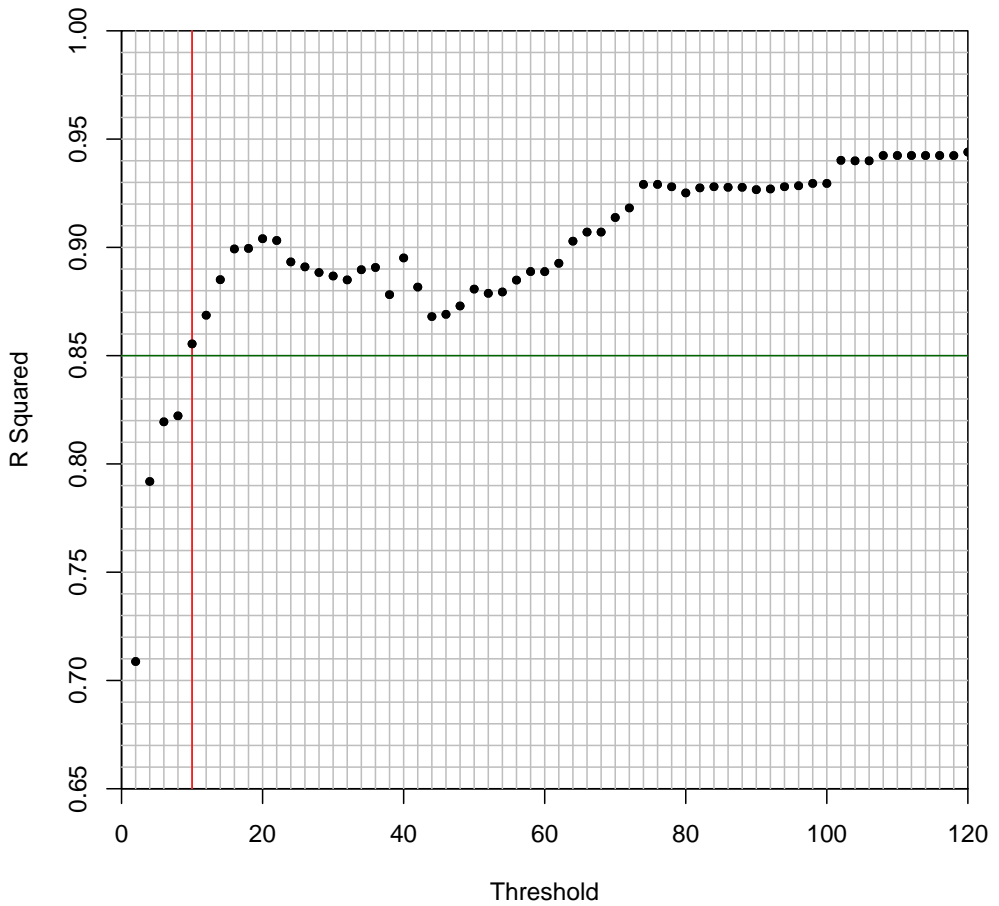


Figure 5.4: R^2 between estimated results and empirical data for different thresholds for second-order mutation. The red line denotes the chosen threshold, and the green line shows the level of R^2 that the chosen threshold guaranties.

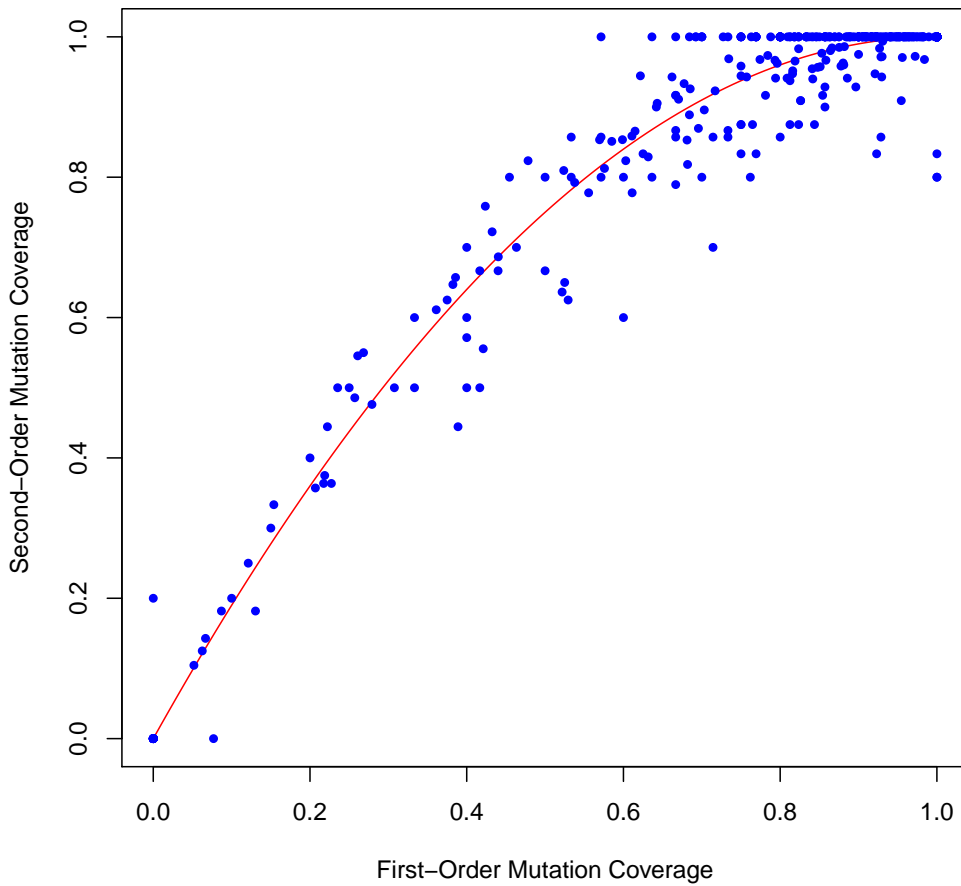


Figure 5.5: Empirical data and the estimated curve for $t = 10$. The red curve is the estimation provided by the model, and each blue dot represents a class.

Approach. We categorize the second-order mutants into three categories: Expected, HK-FS, and HS-FK. Then we compute how many mutants are in each category. The higher the number in the Expected category, the better the model describes the behavior of the second-order mutants.

Findings. Table 5.4 shows the number of higher-order mutants in each category for each project. Out of 10,153 second-order mutants generated for all projects, 10,069 (99.17%) are in the Expected category. This means that overall only 84 (0.83%) mutants are of unexpected status, of which 13 (0.13%) mutants belong to the HK-FS category, and 71 (0.70%) mutants belong to the HS-FK category. In total, 764 (91.28%) out of 837 classes do not contain any HS-FK or HK-FS mutants. In almost all projects the number of mutants in the Expected category is higher than 97%. The only exception is given by the project *ScribeJava*. By manually checking this case, we found that the unexpected second-order mutants were created in very small classes where the underlying first-order mutants directly interact with each other. Overall, we see that the modeled behavior applies to the vast majority of the second-order mutants.

Our model provides a good accuracy in estimating the first-order mutation coverage based on second-order mutation coverage. Moreover, it provides a curve which is close to the best one fitting the empirical data. Finally, the behavior of the vast majority of the second-order mutants is modeled correctly.

5.5.0 RQ2. How Does Incrementing the Order of Mutation Affect the Accuracy of the Estimation Provided By the Model?

Motivation. RQ1 provides evidence of the soundness of the model and shows that second-order mutants can be used by the model to provide good estimations. In this RQ, we are interested in extending this analysis to verify how the higher orders of mutation affects the estimations. Taking into account the trade-off between accuracy of the estimation and computational time required, this RQ helps testers to decide which order of mutation better fits their needs.

Approach. For all projects, we perform third-, fourth-, fifth-, sixth-, and eighth-order mutation testing. Then, we observe how the order of mutation affects the accuracy of the estimations for various thresholds. We also verify whether the model describes the real behavior by computing the number of mutants in the Expected category for the aforementioned orders of mutation.

Findings. Figure 5.6 shows the computed R^2 between estimated results and empirical data for various thresholds t for second, third, fourth, fifth, sixth and eighth orders of

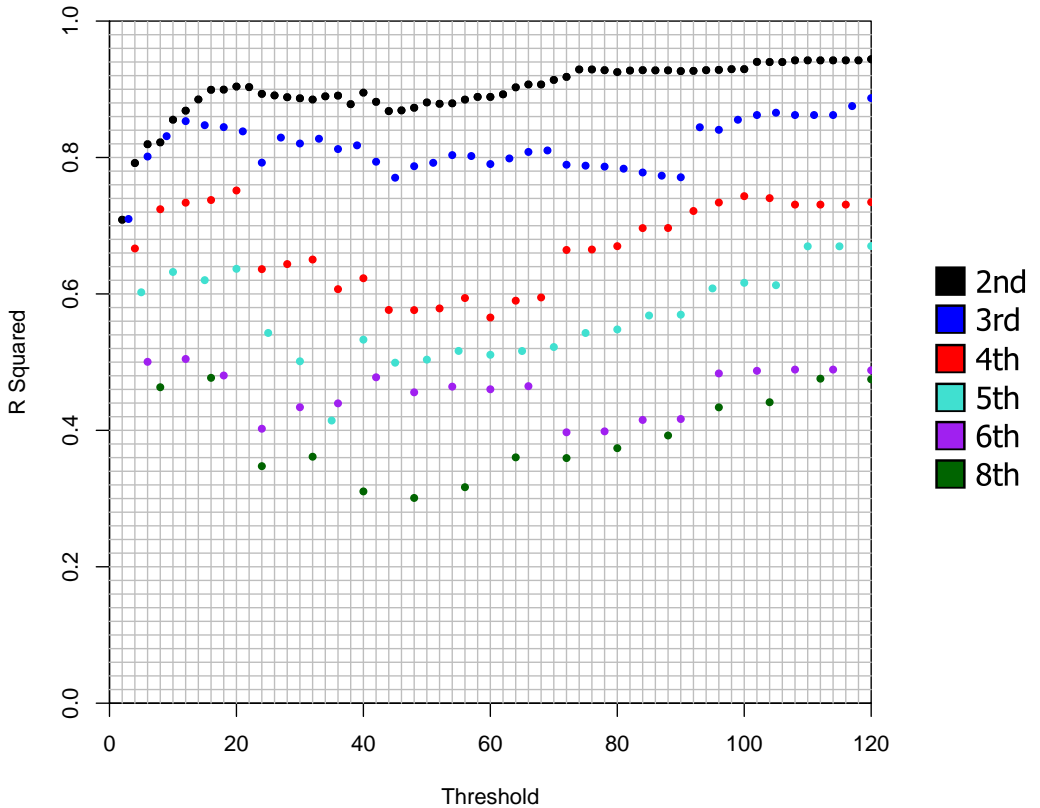


Figure 5.6: R^2 between estimated results and empirical data for different thresholds.

mutation. In this figure, we observe that with the increment of the order of mutation, the R^2 values decrease for any threshold. This can be attributed to the fact that the higher the order of the mutant is, the higher is the chance that at least one of the underlying first-order mutants is killed. This means that increasing the order of mutation makes the model less accurate. From the figure we notice that the values of R^2 remain similar for the second-order and the third-order within $2 \leq t \leq 12$. Whereas, the values of R^2 for fourth and higher orders are visibly lower, highlighting the limitation of proposed model.

As seen in Table 5.4, the percentage of the mutants in the Expected category generally increases along with the order of mutation. This is in accordance to the results of the previous research [112, 115] on coupling effect, confirming the fact that the higher the order of mutation, the lesser the chance of unexpected behavior. The same effect can symmetrically be seen focusing on the decreasing number of mutants in the categories HS-FK

and HK-FS. For HS-FK, there is lesser chance of survival for the higher-order mutant as the order of mutation increases. For HK-FS, there is lesser chance that all the underlying first-order mutants have survived when creating higher-order mutants randomly.

The estimations of the model have an accuracy that decreases with the increment of the order of mutation. This decay is prominent from the fourth order of mutation onwards.

5.6 THREATS TO VALIDITY

To describe the threats to validity we refer to the guidelines reported by Yin [105].

Threats to **internal validity** focus on confounding factors that can influence the obtained results. This threat stems from potential bugs hidden inside the algorithms used for creating higher-order mutants or in LittleDarwin. We consider this chance—even if possible—limited. The results of the experiments are available, and several iterations of post-analysis were performed to make sure of the correctness of the original algorithms. In addition, the code of LittleDarwin has been already checked and tested in several case studies [33, 78, 89].

Threats to **external validity** refers to the generalizability of our results. In our analysis we use only 17 open-source projects. We mitigate this threat by using projects which differ for number of contributors, size and adequacy of the test suite. Yet, it is desirable to replicate this study using more projects, especially the ones belonging to industrial settings. Another threat to external validity arises from the overfitting of the model to the data. Since our model is developed independent of the subjects of the study, we believe this threat does not apply.

Threats to **construct validity** are concerned with how accurately the observations describe the phenomena of interest. In our case, the techniques adopted to evaluate the accuracy of the estimations are: (i) calculation of coefficient of determination (R^2) between the estimated results and empirical data, and (ii) polynomial regression analysis. Both of these techniques are well known, and they have been used in literature numerous times for similar purposes. Another threat to construct validity stems from the fact that the process of creating higher-order mutants requires to combine several first-order mutants selected at random within a class. Therefore, multiple analysis would be needed to reduce the random effect. However, this is not possible, since mutation testing is a time-consuming procedure, and it is not possible to perform all experiments several times. Using R^2 for a data set of 837 extracted classes alleviates this issue to a certain extent by reducing the random effect over the whole dataset. We also use multiple cut-off thresh-

olds to remove outliers from the dataset.

The quality of the mutants affects the results of our study in two ways: First, existence of equivalent first-order mutants reduces the accuracy of the first-order mutation coverage. However, we did not filter for the equivalent mutants, since when combined with a non-equivalent mutant, they do not affect the status of the created higher-order mutant. In other words, the higher-order mutant created from the combination of a killed first-order mutant and an equivalent one is still killed. The chance of a higher-order mutant consisting only out of equivalent mutants decreases exponentially by the order of mutation²¹. But, in general, the detection of equivalent mutants is an undecidable problem [31]. Second, Amman et al. [71] show that large portions of first-order mutants are redundant for all practical purposes. This means that without adequate filtering of the first-order mutants, the resulting mutation coverage does not measure the quality of the test-suite accurately. Since this threat affects both first-order and second-order mutants, the extent of the problem for our empirical validation remains unknown, and requires further investigation.

5.7 RELATED WORKS

Reduction of the number of mutants has been investigated in the literature to reduce the computational cost of mutation testing. Higher-order mutation is first introduced by Offutt in 1992 [112] to investigate the coupling effect empirically. Jia and Harman provide a technique to create “hard to kill” higher-order mutants [68]. Polo et al. [116] and Papadakis et al. [83] evaluated second-order mutation testing by combining first-order mutants using different algorithms and concluded that second-order mutation testing reduces the effort significantly while also reducing the number of equivalent mutants.

Differently from these studies, our study focuses on the accuracy to the estimation of mutation coverage rather than on the creation of hard to kill mutants. In contrast to previous studies where such mutants were desirable, in our case they represent a problem since they make the estimation of first-order mutation coverage less accurate in our model (we assume the faults behave as expected by the coupling effect when combined). In this sense, our work is more geared towards development environments that require an accurate metric to quantify the quality of their test suites.

Kintis et al. [117] shows that using only disjoint second-order mutants creates a more robust measure of test suite effectiveness than first-order mutation coverage. Our model can be used in combination with their method of creating second-order mutants, however, an empirical evaluation of such combination remains as future work.

²¹For example, if 10% of the first-order mutants are equivalent, the chance of a second-order mutant created by the combination of two equivalent mutants is close to 1%.

There are several studies that attempt to validate the coupling effect hypothesis both theoretically and empirically. Offutt provided experiments in support of the hypothesis showing that the vast majority of the higher-order mutants are coupled to the first-order mutants [112, 118]. Wah designed a mathematical model to describe the behavior of the faults in a program consisting only of functions [115, 119]. He showed that by increasing the order of the mutation, the number of de-coupled mutants decrease. Our study agrees with the aforementioned studies, as we show that the number of higher-order mutants with unexpected behavior is very small, and decreases when the order of mutation increases.

5.8 CONCLUSION

First-order mutation coverage is an effective metric to evaluate the quality of the test suite. However, its adoption is hindered due to the high computational time consumption. On one hand, the adoption of higher-order mutants to evaluate the quality of the test suite is a viable solution. On the other hand, this approach provides less realistic estimations than the ones obtained with first-order mutants. As a solution, we propose a realistic model able to estimate first-order mutation coverage based on higher-order mutation coverage. The benefits of this model are that (i) we achieve a good accuracy in estimating the first-order mutation coverage based on second- and third-order mutation coverage, (ii) we halve the computational time that would be required by first-order mutation testing (at minimum). Moreover, the chance of existence of equivalent mutants is less than first-order analysis, even though the model does not explicitly consider them in its estimation. Finally, the model correctly describes the real behavior of vast majority of higher-order mutants.

Dynamic Mutant Subsumption Analysis using LittleDarwin



Dynamic Mutant Subsumption Analysis Using LittleDarwin

Ali Parsai and Serge Demeyer

In *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing (A-TEST 2017)*, 1–4. Paderborn, Germany. September, 2017.

URL: <https://doi.org/10.1145/3121245.3121249>.

This chapter was originally published in the *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing (A-TEST 2017)*.

CONTEXT

This chapter targets the first of our three identified problems, the performance problem. In particular, it aims to describe dynamic mutant subsumption in LittleDarwin, that detects redundancy in a set of mutants. This allows for more reliable mutation coverage, and the possibility to remove redundant mutants from subsequent analyses to save time.

We added more detail about the concept of true subsumption compared to the published version.

ABSTRACT

Many academic studies rely on mutation testing to use as their comparison criteria. However, recent studies have shown that redundant mutants have a significant effect on the accuracy of their results. One solution to this problem is to use mutant subsumption to detect redundant mutants. Therefore, in order to facilitate research in this field, a mutation testing tool that is capable of detecting redundant mutants is needed. In this paper, we describe how we improved our tool, LittleDarwin, to fulfill this requirement.

6.1 INTRODUCTION

Many academic studies on fault detection need to assess the quality of their technique using seeded faults. One of the widely-used systematic ways to introduce faults into the programs is mutation testing [38]. Mutation testing is the process of injecting faults into software (i.e. creating a mutant), and counting the number of these faults that make at least one test fail (i.e. kill the mutant). The process of creating a mutant consists of applying a predefined transformation on the code (i.e. mutation operator) that converts a healthy version of the code into a faulty version. It has been shown that mutation testing is an appropriate method to simulate real faults and perform comparative analysis on testing techniques [24, 81, 120].

There has been many studies to optimize the process of mutation testing by following the maxim *{do faster, do smarter, do fewer}* [30]. In particular, *do fewer* aims to reduce the number of produced mutants by removing the redundant ones. There are several techniques that implement this logic (e.g. selective mutation [49, 101, 106], and mutant sampling [33, 63, 64, 65]). However, only recently the academics began to investigate the threats of validity the redundant mutants introduce in software testing experiments [72]. Papadakis et al. demonstrate that the existence of redundant mutants introduces a significant threat by “artificially inflating the apparent ability of a test technique to detect faults” [72].

One of the recent solutions to alleviate this problem is to use mutant subsumption [71]. Mutant *A* *truly* subsumes mutant *B* if and only if every input data that kills *A* also kills *B*. This means that mutant *B* is redundant, since killing *A* is sufficient to know that *B* is also killed. It is possible to provide a more accurate analysis of a testing experiment by determining and discarding the redundant mutants. However, it is often impossible to check mutants for every possible input to the program in practice. Therefore, as a compromise, dynamic mutant subsumption is used instead [71]. Mutant *A* *dynamically* subsumes mutant *B* with regards to test set *T* if and only if every test that kills *A* also kills *B*. Given the fact that mutant subsumption only recently has been at the center of attention, there are no mature tools that can perform dynamic mutant subsumption analysis

on real-life Java programs. This, however, is necessary to facilitate further research on the topic. Therefore we aim to fill this void by developing such tool.

We used LittleDarwin¹ mutation testing framework to implement the features needed to perform dynamic mutant subsumption analysis. LittleDarwin is an extensible and easy to deploy mutation testing tool for Java programs [32]. LittleDarwin has been used previously in several other studies [33, 34, 78], and it is shown to be capable of analyzing large and complicated Java software systems [89].

The rest of the paper is organized as follows: In Section 6.2, background information about mutation testing is provided. In Section 6.3, the current state of the art is discussed. In Section 6.4, we provide details on how LittleDarwin can help performing dynamic mutant subsumption analysis. Finally, we present our conclusions in Section 6.5.

6.2 BACKGROUND

Mutation testing is the process of injecting faults into a software system to verify whether the test suite detects the injected fault. The idea of mutation testing was first mentioned by Lipton, and later developed by DeMillo, Lipton and Sayward [38]. The first implementation of a mutation testing tool was done by Timothy Budd in 1980 [39]. Mutation testing starts with a *green* test suite — a test suite in which all the tests pass. First, a faulty version of the software is created by introducing faults into the system (*Mutation*). This is done by applying a known transformation (*Mutation Operator*) on a certain part of the code. After generating the faulty version of the software (*Mutant*), it is passed onto the test suite. If there is an error or failure during the execution of the test suite, the mutant is marked as killed (*Killed Mutant*). If all tests pass, it means that the test suite could not catch the fault, and the mutant has survived (*Survived Mutant*) [31].

If the output of a mutant for all possible input values is the same as the original program, it is called an *equivalent mutant*. It is not possible to create a test case that passes for the original program and fails for an equivalent mutant, because the equivalent mutant is indistinguishable from the original program. This makes the creation of equivalent mutants undesirable, and leads to false positives during mutation testing. In general, detection of equivalent mutants is undecidable due to the halting problem [42]. Manual inspection of all mutants is the only way of filtering all equivalent mutants, which is impractical in real projects due to the amount of work it requires.

$$\text{Mutation Coverage} = \frac{\text{Number of killed mutants}}{\text{Number of all non-equivalent mutants}} \quad (6.1)$$

Mutation testing allows software engineers to monitor the fault detection capability

¹<https://littledarwin.parsai.net/>

of a test suite by means of mutation coverage (see Equation 6.1) [31]. A test suite is said to achieve *full mutation test adequacy* whenever it can kill all the non-equivalent mutants, thus reaching a mutation coverage of 100%. Such test suite is called a *mutation-adequate test suite*.

6.3 STATE OF THE ART

Mutant subsumption is defined as the relationship between two mutants A and B in which A subsumes B if and only if the set of inputs that kill A is guaranteed to kill B [69]. The extraction of the true subsumption relationships between all mutant is a generally undecidable problem. The subsumption relationship for faults has been defined by Kuhn in 1999 [70], but its use for mutation testing has been popularized by Jia et al. for creating hard to kill higher-order mutants [67]. Later on, Ammann et al. tackled the theoretical side of mutant subsumption [71]. In their paper, Ammann et al. define *dynamic* mutant subsumption, which redefines the relationship using test cases. Mutant A dynamically subsumes Mutant B if and only if (i) A is killed, and (ii) every test that kills A also kills B. Kurtz et al. [69] use the notion of dynamic mutant subsumption graph (DMSG) to visualize the concept of dynamic mutant subsumption. Each node in a DMSG represents a set of all mutants that are mutually subsuming. Edges in a DMSG represent the dynamic subsumption relationship between the nodes. They introduce the concept of static mutant subsumption graph, which is a result of determining the subsumption relationship between mutants using static analysis techniques. The main purpose behind the use of mutant subsumption is to reliably detect redundant mutants, which create multiple threats to the validity of mutation testing [72]. This is often done by determining the dynamic subsumption relationship among a set of mutants, and keeping only those that are not subsumed by any other mutant.

6.4 DYNAMIC MUTANT SUBSUMPTION ANALYSIS WITH LITLEDARWIN

Figure 6.1 shows the input and output of LittleDarwin’s dynamic mutant subsumption (DMS) component. To facilitate dynamic mutant subsumption analysis in LittleDarwin, we retain all the output provided by the build system for each mutant. As a result, we can parse this output and extract useful information, e.g. which test cases fail for a particular mutant. LittleDarwin’s DMS component can then use this information to determine dynamic subsumption relation between each mutant pair. This component then outputs the results in two different ways: (i) the dynamic mutant subsumption graph, to visualize the subsumption relation, and (ii) a detailed report is generated in CSV² format

²Comma-separated Values

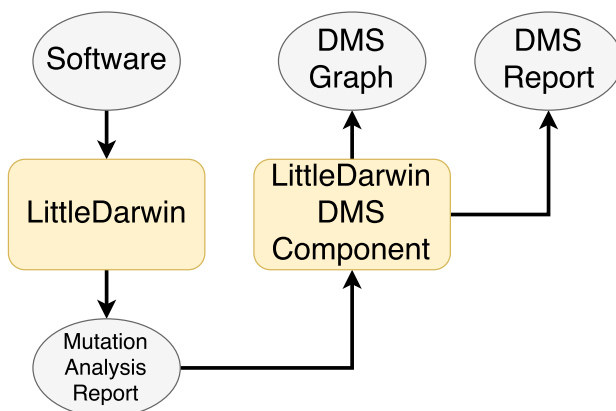


Figure 6.1: Dynamic Mutant Subsumption Component I/O

Table 6.1: JTerminal Software Information

Project	Ver.	Size (LoC)		#C	TS	SC	BC	MC	#M
		Prod.	Test						
JTerminal	1.0.1	687	250	8	2	66%	56%	60.0%	160

Acronyms: Version (Ver.), Line of code (LoC), Production code (Prod.), Number of commits (#C), Team size (TS), Statement coverage (SC), Branch coverage (BC), Mutation coverage (MC), Number of Mutants (#M)

that contains all the information processed by the DMS component. For each mutant, mutant ID, mutant path, source path, mutated line number, whether it is a subsuming mutant, number of failed tests, the mutants it subsumes, the mutants that it is subsumed by, and the mutants that are mutually subsuming with it are provided in this report.

To showcase the ability of LittleDarwin in performing dynamic mutant subsumption analysis, we use JTerminal³ as a subject. The information about characteristics of JTerminal is shown in Table 6.1. The DMSG for JTerminal is depicted in Figure 6.2. In this figure, each number represents a single killed mutant, each node represents a group of mutants that are killed by exactly the same set of test cases, and each edge shows the dynamic subsumption relationship between each node where the node at the end is subsumed by the node at the start. The survived mutants and the equivalent mutants are not shown in this figure. The double-circled nodes contain the subsuming mutant groups. In order to remove the redundant mutants, one only needs to keep one mutant from each subsuming mutant group and discard the rest.

Take Mutants 72, 88, and 111 as an example. According to the DMSG, Mutants 88

³<https://www.grahamedgecombe.com/projects/jterminal>

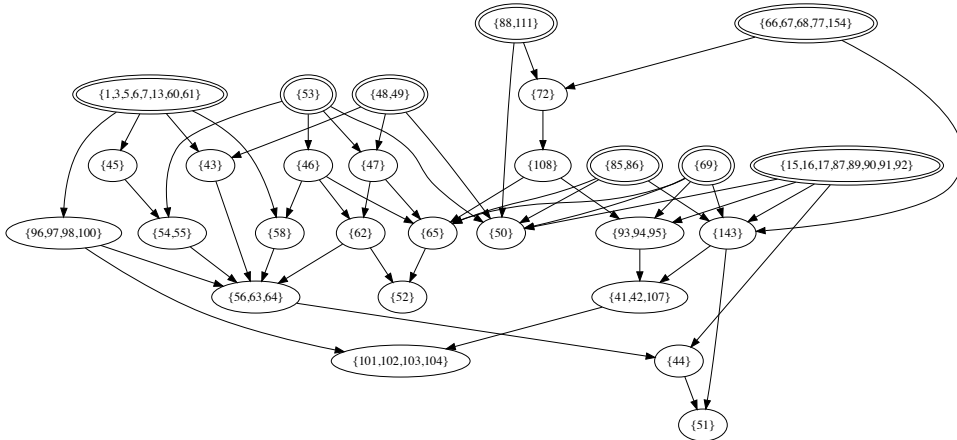


Figure 6.2: Dynamic Mutant Subsumption Graph for JTerminal

```

case '\n':
    cursorColumn = 0;
    cursorRow++; /* Mutant 72 -> cursorRow--; */
    continue;
case '\t':
    while ((++cursorColumn % TAB_WIDTH) != 0);
    /* Mutant 88 -> while ((++cursorColumn / TAB_WIDTH) != 0); */
    /* Mutant 111 -> while ((++cursorColumn % TAB_WIDTH) == 0); */
    continue;

```

Figure 6.3: Mutants 72, 88, and 111 of JTerminal

and 111 subsume Mutant 72, and Mutants 88 and 111 are mutually subsuming. Using the CSV report, we can locate the actual mutation of the source code. Mutants 72, 88, and 111 belong to method `parsedString` of class `Vt100TerminalModel`. The relevant part of the code is shown in Figure 6.3. Since Mutant 72 is applied to a more critical branch, it triggers an additional test case failure compared to Mutants 88 and 111, hence it is subsumed by them. Mutants 88 and 111 are affecting the same statement in different ways. Mutant 88 is more likely to make the loop iterate more than it should, while Mutant 111 is more likely to terminate it instantly. Therefore, in both cases the same loop is affected, and the same test case fails. Please note that Mutants 88 and 111 are only *dynamically* mutually subsuming, meaning that given the current test suite there is no distinction between them, even though it is theoretically possible to create a test case that distinguishes between the two.

Analysis such as this allows researchers to understand the relations between the mutants and reduce the effects of redundant mutants on their results. This would also help industrial adoption by reducing the load of consecutive runs of mutation testing. Once the set of redundant mutants are identified for a revision, the consecutive runs can remove those from the analysis. Given the fact that often up to 70% of mutants can be redundant [72], this offers a large saving in the overall performance of the process.

6.5 CONCLUSION

Many academic studies rely on mutation testing to use as their comparison criteria, and the existence of redundant mutants is a significant threat to their validity. We developed a component for our mutation testing tool, LittleDarwin, to facilitate the detection of redundant mutants using dynamic mutant subsumption analysis. We performed such analysis on a small, real-world project to demonstrate the capabilities of our tool. Using our tool, it is possible to detect and filter out redundant mutants, and help in increasing the confidence in the results of experiments using mutation testing as a comparison criteria.

Do Null-Type Mutation Operators Help Prevent Null-Type Faults?



Do Null-Type Mutation Operators Help Prevent Null-Type Faults?

Ali Parsai and Serge Demeyer

In *SOFSEM 2019: Theory and Practice of Computer Science (SofSem 2019)*, 419–434..
January, 2019.

URL: https://doi.org/10.1007/978-3-030-10801-4_33.

This chapter was originally published in the *SOFSEM 2019: Theory and Practice of Computer Science (SofSem 2019)*.

CONTEXT

This chapter targets the second of our three identified problems, the fault model problem. In particular, it aims to introduce new mutation operators that target the null-type faults in Java. Null-type faults are a source of major concern in Java programs, and there are several patterns of such faults identified in the academic literature. We build on top of these patterns to introduce new mutation operators that explicitly model these faults, and allow developers to write tests targeting null-type faults.

We added a clarification about invalid mutants and extended the threats to validity section compared to the originally published paper.

ABSTRACT

The null-type is a major source of faults in Java programs, and its overuse has a severe impact on software maintenance. Unfortunately traditional mutation testing operators do not cover null-type faults by default, hence cannot be used as a preventive measure. We address this problem by designing four new mutation operators which model null-type faults explicitly. We show how these mutation operators are capable of revealing the missing tests, and we demonstrate that these mutation operators are useful in practice. For the latter, we analyze the test suites of 15 open-source projects to describe the trade-offs related to the adoption of these operators to strengthen the test suite.

7.1 INTRODUCTION

The null-type is a special type in Java that has no name, cannot be casted, and practically equates to a literal that can be of any reference type [121]. The null-type is commonly misused, and frequently reported and discussed as an issue by developers [122]. The null-type is the source of the majority of faults in Java programs [92], and its overuse has a severe impact on software maintenance [123]. A null-type fault is a fault that is caused by mishandling of null, often leading to an uncaught null pointer exception. On the one hand, this scenario should push developers to build test suites capable of identifying null-type faults. On the other hand, developers without specific test requirements may struggle to identify all code elements or properties that the test must satisfy. To address this problem, we propose mutation testing as a way for improving the test suite to handle potential null-type faults.

Mutation testing is a technique to measure the quality of a test suite by assessing its fault detection capabilities [38]. Mutation testing is a two-step process. First, a small syntactic change is introduced in the production code. This change is obtained by applying a “mutation operator”, and the resulting changed code is called a “mutant”. Then, the test suite is executed for that mutant; if any of the tests fail, the mutant is “killed”, otherwise, the mutant has “survived”. Herein lies the aspect of mutation testing that we want to exploit: the identification of survived mutants that need to be killed. Mutation operators are modeled after the common developer mistakes [81]. Over the years, multiple sets of mutation operators have been created to fit in different domains. By far the most commonly used mutation operators are the ones introduced in Mothra by Offutt et al. [49]. They use 10 programs written in Fortran to demonstrate that their reduced-set mutation operators is enough to produce a mutation-adequate test suite that can kill almost all of the mutants generated by the mutation operators of the complete-set. Later on, several attempts have been made to extend Offutt’s mutation operators, for instance, to cope with the specificities of object-oriented programming [50]. Yet, none of the proposed mutation

operators explicitly model null-type faults. As a result, mature general-purpose mutation testing tools currently used in literature, such as PITest [75] and Javalanche [73], do not cope explicitly with this type of faults by default. Therefore, the created mutants *risk* not being adequate to derive test requirements that handle null-type faults. Whether this risk is concrete or not depends on the ability of the available mutation operators to account for these faults. Yet, no study has explored this aspect.

This paper investigates the usefulness of mutation operators able to model null-type faults in order to strengthen the test suite against these faults. For this reason, we introduce four new mutation operators related to null-type faults. These mutation operators are modeled to cover the typical null-type faults introduced by developers [122]. We incorporate these mutation operators in LittleDarwin, an extensible open-source tool for mutation testing [32], creating a new version called LittleDarwin-Null. We organize our research in two steps: we show that (i) the current general-purpose mutation testing tools do not account for null-type faults by default, and modeling operators for null-type faults can drive the improvement of the test suite in practice, and (ii) the test suites of real open-source projects cannot properly catch null-type faults. The paper is driven by the following research questions:

- **RQ1:** *Are traditional mutation operators enough to prevent null-type faults?*
- **RQ2:** *To what extent is the addition of null-type mutation operators useful in practice?*

The rest of the paper is organized as follows: In Section 7.2, background information and related work is provided. In Section 7.3, the details of the experiment are discussed. In Section 7.4, the results are analyzed. In Section 7.5, we discuss the threats that affect the results. Finally, we present the conclusion in Section 7.6.

7.2 BACKGROUND AND RELATED WORK

Mutation testing is the process of injecting faults into a software system and then verifying whether the test suite indeed fails, and thus detects the injected fault. First, a faulty version of the software is created by introducing faults into the system (*Mutation*). This is done by applying a transformation (*Mutation Operator*) on a certain part of the code. After generating the faulty version of the software (*Mutant*), it is passed onto the test suite. If a test fails, the mutant is marked as killed (*Killed Mutant*). If all tests pass, the mutant is marked as survived (*Survived Mutant*).

Mutation Operators. A mutation operator is a transformation which introduces a single syntactic change into its input. The first set of mutation operators were reported in King et al. [48]. These mutation operators work on essential syntactic entities of programming languages such as arithmetic, logical, and relational operators. For object-oriented

languages, new mutation operators were proposed [50]. The mature mutation testing tools of today still mostly use the *traditional* (i.e. method-level) mutation operators [23].

Equivalent Mutants. An *equivalent mutant* is a mutant that does not change the semantics of the program, i.e. its output is the same as the original program for any possible input. Therefore, no test case can differentiate between an equivalent mutant and the original program. The detection of equivalent mutants is undecidable due to the halting problem [42]. **Mutation Coverage.** Mutation testing allows software engineers to monitor the fault detection capability of a test suite by means of mutation coverage [31]. A test suite is said to achieve *full mutation test adequacy* whenever it can kill all the non-equivalent mutants, thus reaching a mutation coverage of 100%. Such test suite is called a *mutation-adequate test suite*.

Mutant Subsumption. Mutant subsumption is defined as the relationship between two mutants A and B in which A subsumes B if and only if the set of inputs that kill A is guaranteed to kill B [69]. The subsumption relationship for faults has been defined by Kuhn in 1999 [70]. Later on, Ammann et al. tackled the theoretical side of mutant subsumption [71] where they define *dynamic* mutant subsumption as follows: Mutant A dynamically subsumes Mutant B if and only if (i) A is killed, and (ii) every test that kills A also kills B. The main purpose behind the use of mutant subsumption is to detect redundant mutants. These mutants create multiple threats to the validity of mutation analysis [72]. This is done by determining the dynamic subsumption relationship among a set of mutants, and keep only those that are not subsumed by any other mutant.

Mutation Testing Tools. In this study, we use three different mutation testing tools: Javalanche, PITest, and LittleDarwin. Javalanche is a mutation testing framework for Java programs that attempts to be efficient, and not produce equivalent mutants [73]. It uses byte code manipulation in order to speed up the process of mutation testing. Javalanche has been used in numerous studies in the past (e.g. [47, 74]). PITest is a state-of-the-art mutation testing system for Java, designed to be fast and scalable [75]. PITest is the de facto standard for mutation testing within Java, and it is used as a baseline in mutation testing research (e.g. [76, 77]). LittleDarwin is a mutation testing tool designed to work out of the box with complicated industrial build systems. For this, it has a loose coupling with the test infrastructure, instead relying on the build system to run the test suite. LittleDarwin has been used in several studies, and is capable of performing mutation testing on complicated software systems [33, 34, 78]. For more information about LittleDarwin please refer to Parsai et al. [32]. We implemented the new null-type mutation operators in a special version of LittleDarwin called LittleDarwin-Null. LittleDarwin and LittleDarwin-Null only differ in the set of mutation operators used, and are identical otherwise.

Related Work. Creating new mutation operators to deal with the evolution of soft-

Table 7.1: Null-Type Faults and Their Corresponding Mutation Operators

Mutation Operator	Description
NullifyReturnValue	If a method returns an object, it is replaced by <code>null</code>
NullifyInputVariable	If a method receives an object reference, it is replaced by <code>null</code>
NullifyObjectInitialization	Wherever there is a <code>new</code> statement, it is replaced with <code>null</code>
NegateNullCheck	Any binary relational statement containing <code>null</code> at one side is negated

ware languages is a trend in mutation testing research. For example, mutation operators have been designed to account for concurrent code [124], aspect-oriented programming [125], graphical user interfaces [126], modern C++ constructs [37], and Android applications [127]. Nanavati et al. have previously studied mutation operators targeting memory-related faults [128]. However, the difference in the semantics of null object of Java and `NULL` macro of C is sufficient to grant the need for a separate investigation.

7.3 EXPERIMENTAL SETUP

In this section, we first introduce our proposed mutation operators, and then we discuss the experimental setup we used to address our research questions.

7.3.1 Null-Type Mutation Operators

We derived four null-type mutation operators to model the typical null-type faults often encountered by developers [92]. These mutation operators are presented in Table 7.1.

7.3.2 Case Study

For RQ1, we use a didactic project. For RQ2, we use 15 open-source projects.

RQ1. In order to address RQ1, we chose a modified version of VideoStore as a small experimental project [129]. Choosing a small project allows us to (i) create a mutation-adequate test suite ourselves, (ii) find out which mutants are equivalent, and (iii) avoid complexities when using multiple mutation testing tools. The source code for VideoStore is available in the replication package.

RQ2. We selected 15 open-source projects for our empirical study (Table 7.2). The selected projects differ in size of their production code and test code, number of commits, and team size to provide a wide range of possible scenarios. Moreover, they also differ in the adequacy of their test suite based on statement, branch, and mutation coverage (Table 7.2). We used JaCoCo and Clover for statement and branch coverage, and LittleDarwin for mutation coverage.

Table 7.2: Projects Sorted by Mutation Coverage

Project	Ver.	Size (LoC)		#C	TS	SC	BC	MC
		Prod.	Test					
Apache Commons CLI	1.3.1	2,665	3,768	816	15	96%	93%	94%
JSQParser	0.9.4	7,342	5,909	576	19	81%	73%	94%
jOpt Simple	4.8	1,982	6,084	297	14	99%	97%	92%
Apache Commons Lang	3.4	24,289	41,758	4,398	30	94%	90%	91%
Joda Time	2.8.1	28,479	54,645	1,909	42	90%	81%	82%
Apache Commons Codec	1.10	6,485	10,782	1,461	10	96%	92%	82%
Apache Commons Collections	4.1	27,914	32,932	2,882	26	85%	78%	81%
VRaptor	3.5.5	14,111	15,496	3,417	65	87%	81%	81%
HTTP Request	6.0	1,391	2,721	446	15	94%	75%	78%
Apache Commons FileUpload	1.3.1	2,408	1,892	846	19	76%	74%	77%
jsoup	1.8.3	10,295	4,538	888	43	82%	72%	76%
JGraphT	0.9.1	13,822	8,180	1,150	31	79%	73%	69%
PITest	1.1.7	17,244	19,005	1,044	19	79%	73%	63%
JFreeChart	1.0.17	95,354	41,238	3,394	4	53%	45%	35%
PMD	r7706	70,767	43,449	7,706	20	62%	54%	34%

Acronyms: Version (Ver.), Line of code (LoC), Production code (Prod.), Number of commits (#C), Team size (TS), Statement coverage (SC), Branch coverage (BC), Mutation coverage (MC)

Table 7.3: Mutation testing results for VideoStore

Program	LittleDarwin			PITest			Javalanche			LittleDarwin-Null		
	K	S	E	K	S	E	K	S	E	K	S	E
VideoStore Orig	24	18	2	25	43	5	87	69	11	11	14	1
VideoStore TAdq	42	0	2	68	0	5	202	0	11	22	3	1
VideoStore NAdq	42	0	2	68	0	5	202	0	11	25	0	1

K: Killed, S: Survived, E: Equivalent

7.4 RESULTS AND DISCUSSION

RQ1: Are traditional mutation operators enough to prevent null-type faults?

We are interested to compute the number of killed, survived and equivalent mutants along with three versions of VideoStore. The first version we analyze is the original one (VideoStore Orig). This version has only 4 tests. Then, we create a mutation-adequate test suite that kills all mutants generated by the general-purpose tools (Javalanche, PITest, and LittleDarwin). In this version (VideoStore TAdq) we added 15 tests. Finally, we create a mutation-adequate test suite that kills all mutants, included the ones generate by LittleDarwin-Null. In this version (VideoStore NAdq) we added 3 more tests.

Table 7.3 shows the number of remaining mutants after each phase of test development: VideoStore Orig, VideoStore TAdq, and VideoStore NAdq. The discrepancy




Mutant A	<pre> public double determineAmount(int daysRented) throws Exception { if (daysRented <= 0) throw new Exception("Invalid value for daysRented."); return 0; } </pre> <p style="text-align: right; margin-right: 50px;">  null </p>
Mutant B	<pre> public Customer(String name) throws NullPointerException { if (name == null) throw new NullPointerException("name is Null"); this.name = name; } </pre> <p style="text-align: right; margin-right: 50px;">  null </p>
Mutant C	<pre> public Rental(Movie movie, int daysRented) { this.movie = movie; if (movie == null) this.movie = new RegularMovie(null); this.daysRented = daysRented; } </pre> <p style="text-align: right; margin-right: 50px;">  null </p>

Figure 7.1: The Surviving Non-Equivalent Null-Type Mutants

in total number of generated mutants for the three versions of the program in case of Javalanche is due to its particular optimizations. In VideoStore Orig, there are several survived mutants according to all the tools. This is because the test suite accompanying the VideoStore program was not adequate.

In VideoStore TAdq, we create a mutation-adequate version of the test suite with respect to the results of PITest, Javalanche, and LittleDarwin. In the process of creating this test suite, we noticed that all of these tools produce equivalent mutants. Two of such mutants are shown in Figure 7.2. Mutant A is equivalent because the method `super.determineAmount` always returns 0, so it does not matter whether it is added to or subtracted from `thisAmount`. Mutant B is also equivalent, because if `daysRented` is 2, the value added to `thisAmount` is 0. We analyzed VideoStore TAdq with LittleDarwin-Null in order to find out whether the mutation-adequate test suite according to three general-purpose tools is able to kill all the null-type mutants. By analyzing the 26 generated mutants, we noticed that 22 mutants were killed and 4 survived. The manual review of these mutants show that one of them is an equivalent mutant.

```

@Override
public double determineAmount(int daysRented) throws Exception {
+ → - 1 double thisAmount = 2 + super.determineAmount(daysRented);
> → >= 2 if (daysRented > 2)
        thisAmount += (daysRented - 2) * 1.5;
    return thisAmount;
}


```

Figure 7.2: Two of the Equivalent Mutants Generated by Traditional Mutation Operators

```

public void addRental(Rental rental) throws NullPointerException {
    if (rental == null)
        rentals.addElement(new Rental(new RegularMovie(null), 0));
    else
        rentals.addElement(rental);
}

```



 null

Figure 7.3: One of the Equivalent Mutants Generated by Null-Type Mutation Operators

Considering that 3 mutants generated by null-type mutation operators are not equivalent, and yet the mutation-adequate test suite we created according to the general-purpose tools cannot kill them, we conclude that **using traditional mutation operators to strengthen the test suite does not necessarily prevent null-type faults.**

The four mutants survived in VideoStore TAdq are all of type *NullifyObjectInitialization*. Figure 7.3 shows the equivalent null-type mutant. Here the default behavior of Rental object is to create a new RegularMovie object when it receives null as its input. So, replacing `new RegularMovie(null)` with `null` does not change the behavior of the program.

The three remaining surviving mutants are described in Figure 7.1. Here, mutants A and B replace the exception with `null`. Consequently, as opposed to the program throwing a detailed exception, the mutant always throws an empty `NullPointerException`. Such a mutant is desirable to kill, since the program would be able to throw an unexpected exception due to a fault that the test suite cannot recognize. In the case of Mutant C, it replaces the initialization of a `RegularMovie` object with `null`. This means that as opposed to the program that guarantees the private attribute `movie` is always instantiated, the same attribute contains a null literal in the mutant. If not detected, a `NullPointerException` might be thrown when another object tries to access the `movie` attribute of this object.

We created three new tests to kill each of the survived mutants. These tests are shown

<pre> @Test public void testMutantA() { movieInstance1 = new ChildrensMovie("null"); movieInstance2 = new RegularMovie("null"); movieInstance3 = new NewReleaseMovie("null"); try { movieInstance1.determineAmount(-8); } catch (Exception e) { assertTrue(e.getMessage().equals("Invalid value for daysRented.")); } try { movieInstance2.determineAmount(-48); } catch (Exception e) { assertTrue(e.getMessage().equals("Invalid value for daysRented.")); } try { movieInstance3.determineAmount(-248); } catch (Exception e) { assertTrue(e.getMessage().equals("Invalid value for daysRented.")); } } </pre>	<h3>Test Killing Mutant A</h3>
<pre> @Test public void testMutantB() { try { new Customer(null); } catch (NullPointerException e) { assertTrue(e.getMessage().equals("name is Null")); } } </pre>	<h3>Test Killing Mutant B</h3>
<pre> @Test public void testMutantC() { Rental rental = new Rental(null, 42); Movie movie = rental.getMovie(); assertTrue(movie instanceof RegularMovie); } </pre>	<h3>Test Killing Mutant C</h3>

Figure 7.4: The Tests Written to Kill the Surviving Null-Type Mutants

in Figure 7.4. Here, `testMutantA` and `testMutantB` verify whether the unit under test throws the correct exception if called with an invalid input value. `testMutantC` verifies whether the unit under test is able to handle a null input correctly. These three tests are not “happy path tests”, namely a well-defined test case using known input, which executes without exception and produces an expected output. Consequently, they might not be intuitive for a test developer to consider, even though they are known as good testing practice [130]. If not for the three survived null-type mutants, these tests would not have been written. This leads us to conclude that **traditional mutation operators are not enough to prevent null-type faults.**

RQ2: To what extent is the addition of null-type mutation operators useful in practice?

RQ1 shows for the VideoStore project that mutation testing tools need to introduce explicit mutation operators for modeling null-type faults. Yet, such a project is not representative of real projects. In this RQ, we want to verify to what extent null-type mutation operators are useful *in practice*. For this reason, we perform an experiment that involves real open-source projects. After introducing null-type mutants, two groups of mutants are affected: (i) survived mutants are the targets the developer needs during test development, (ii) killed mutants show the types of faults the test suite can already catch.

Considering this, we can justify the effort needed for extending mutation testing by incorporating null-type mutants only if: (i) the real test suites do not already kill most of the null-type mutants, (ii) the null-type mutants are not increasing redundancy by a large margin. Otherwise, the current mutation testing tools are already “good enough” for preventing null-type faults.

To verify to what extent the null-type mutants “do matter” when testing for null-type faults we analyze both killed and survived mutants:

In case of survived mutants, we analyze the number of survived mutants that each mutation operator generates for each project. We divide this analysis into two parts. First, we analyze survived mutants for null-type and traditional mutation operators. Second, we analyze each mutation operator individually to find out which one produces the most surviving mutants. This analysis shows whether the survived mutants produced by the null-type mutation operators are “enough” to drive the test development process.

In case of killed mutants, we take all projects as a whole, and we analyze whether the killed null-type mutants are redundant when used together with traditional mutation operators. We measure redundancy using dynamic mutant subsumption: we analyze the distributions of subsuming, killed, and all null-type mutants. This way we can tell whether or not the null-type mutation operators are producing “valuable” mutants to strengthen the test suite.

Survived mutants. Table 7.4 shows for each project the number of survived, killed, and

Table 7.4: Mutants Generated by LittleDarwin and LittleDarwin-Null

Project	Traditional Mutation Operators			Null-Type Mutation Operators		
	Survived	Killed	Total	Survived	Killed	Total
Apache Commons CLI	24	318	342	71	415	486
JSQParser	31	457	488	358	1,062	1,420
jOpt Simple	17	189	206	37	494	531
Apache Commons Lang	559	5,455	6,014	564	5,469	6,033
Joda Time	892	3,978	4,870	836	5,371	6,207
Apache Commons Codec	364	1,612	1,976	147	927	1,074
Apache Commons Collections	638	2,705	3,343	1,179	5,851	7,030
VRaptor	111	478	589	795	2,111	2,906
HTTP Request	49	178	227	69	383	452
Apache Commons FileUpload	81	273	354	137	211	348
jsoup	291	928	1,219	553	1,455	2,008
JGraphT	416	940	1,356	834	1,457	2,291
PITest	398	672	1,070	551	2,964	3,515
JFreeChart	10,558	5,603	16,161	8,563	6,248	14,811
PMD	5,205	2,734	7,939	5,099	4,613	9,712
Total	19,634	26,520	46,154	19,793	39,031	58,824

total generated mutants for both groups of mutation operators. The first noticeable trend is a strong correlation ($R^2 = 0.81$) between survived to killed ratio (SKR) of the traditional mutants and SKR of the null-type mutants. One exception to this trend is JSQParser, in which there are significantly more survived null-type mutants than survived traditional mutants. Investigating further, we find that this happens because 50 small classes lack statements that can be mutated by the traditional mutation operators. However, null-type mutation operators are able to generate mutants for these classes. This uncovers many of the weaknesses of the test suite. On the other side of the fence, there is PITest, in which a single class (`sun.pitest.CodeCoverageStore`) contains many arithmetic operations while poorly tested, so it produces 129 out of 398 survived traditional mutants. This shows that **the usefulness of the null-type mutation operators is program-dependent**.

Figure 7.5 shows the number of killed and survived mutants for each mutation operator before the removal of invalid mutants. We see that among the traditional mutation operators, *ArithmeticOperatorReplacementBinary*, *LogicalOperatorReplacement*, and *ArithmeticOperatorReplacementUnary* have the highest ratio of survived to killed mutants. This means that these mutation operators are generating mutants that are harder to kill than the rest. The same can be observed among the null-type mutation operators, where *NullifyObjectInitialization* produces harder to kill mutants than the others. This is as we expected, since *NullifyInputVariable* applies a major change to the method (removal of an input), and *NegateNullCheck* negates a check that the developer deemed necessary. However, the unexpected part of the result is that so many of the mutants generated by *NullifyReturnValue* have survived. This means that lots of methods are not tested on their

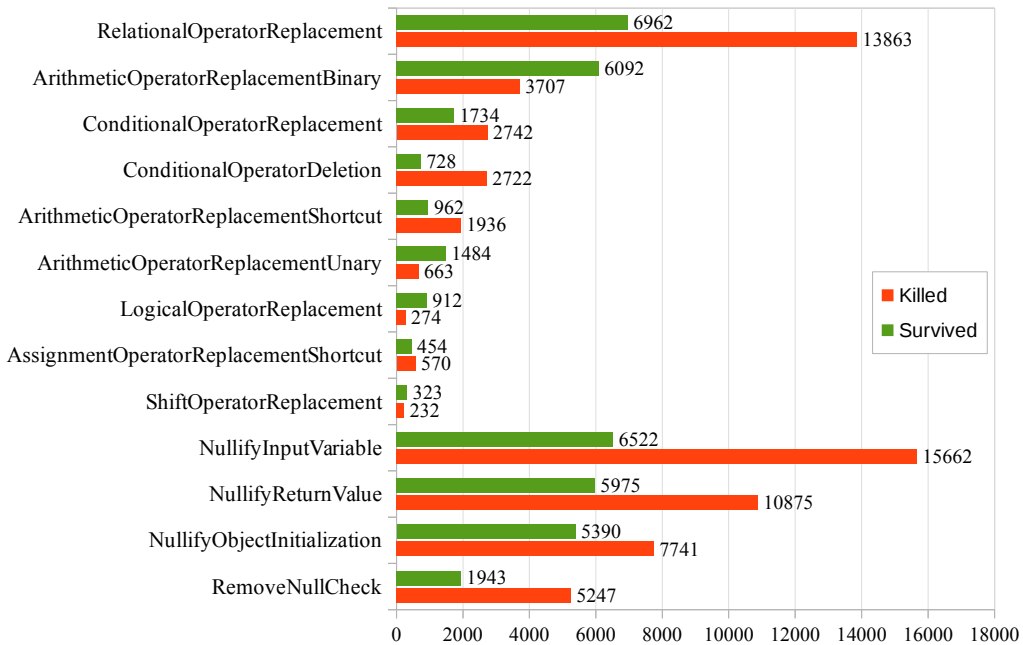


Figure 7.5: Number of killed and survived mutants for each mutation operator

output correctly. This can be due to the fact that many of such methods are not tested directly, and when tested indirectly, their results only affect a small part of the program state of the method under test.

In general, the number of survived null-type mutants has a strong correlation with the number of survived traditional mutants for most projects. This implies that not all parts of the code are tested well. However, the exceptions to this rule are caused by classes that produce many more mutants of a particular type. Here, our results show that **the null-type mutation operators complement the traditional mutation operators and vice versa by each providing a large portion of survived mutants.**

Killed mutants. Considering all projects as a whole and after the removal of 737 invalid mutants, the number of valid generated mutants is 104,978. Out of this total, the number of killed and subsuming mutants are 65,551 and 16,205 respectively. This means that at least 50,029 were subsumed, and thus redundant. To put null-type and traditional mutants in perspective, Figure 7.6 shows the percentages for all, killed, and subsuming mutants for both groups. Here, we notice that the percentage of the null-type mutants remains similar in these three categories. The null-type mutants have a higher impact on the semantics of the program due to being applied at the entry and exit points of a method, the branching statements, and the declaration of an object. Therefore, the fact

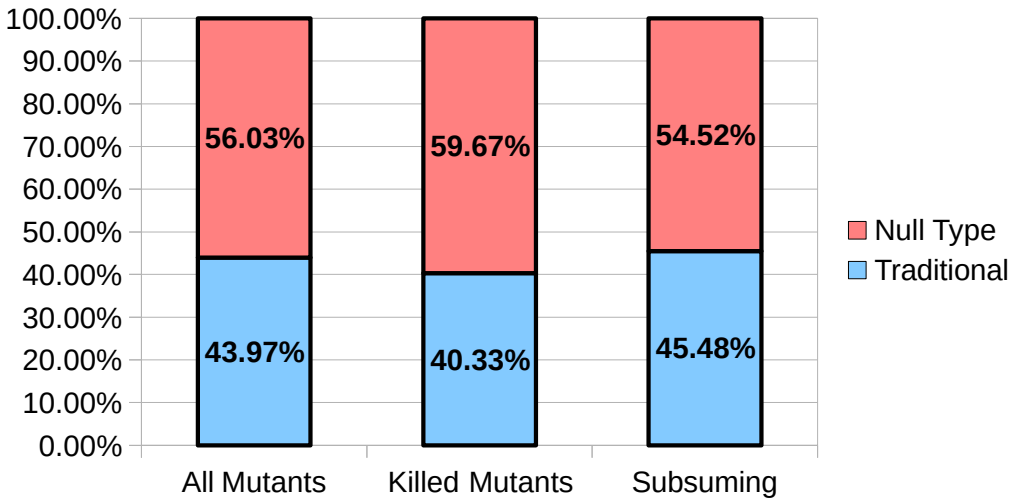


Figure 7.6: Ratio of Null-Type and Traditional Mutants in All, Killed, and Subsuming

that they comprise a higher percentage of the killed mutants is not surprising. This does not reduce the value of such mutants, since it has been shown that even high-impact mutants can detect weaknesses in a test suite, e.g. in case of pseudo-tested methods [131]. It is important to note that the distribution of null-type mutants differs only 4% in all and killed mutants. While 60% of the killed mutants are null-type, they still account for almost 55% of subsuming mutants. This indicates that **the inclusion of the null-type mutants increases the mutant redundancy only marginally.**

To delve deeper, Figure 7.7 shows for each mutation operator the percentage of killed and subsuming mutants. Among the traditional mutation operators, *RelationalOperatorReplacement* and *ConditionalOperatorReplacement* produce the most subsuming mutants. The rest of the mutation operators create mutants that have the same distribution among subsuming and killed mutants. As this figure shows, the marginal increase in redundancy by the null-type mutation operators can be blamed on *NullifyInputVariable* mutation operator. This mutation operator produces mutants that are easier to kill compared to other mutation operators (21% of all, 24% of the killed), and more of these mutants are redundant compared to others (24% of killed, only 15% of subsuming). On the contrary, *NullifyReturnValue* is producing fewer redundant mutants, which confirms our previous observation.

Given the results of RQ2, we conclude that **while the inclusion of the null-type mutation operators increases the redundancy marginally, they complement the traditional mutation operators in their role of strengthening the test suite against null-type faults.**

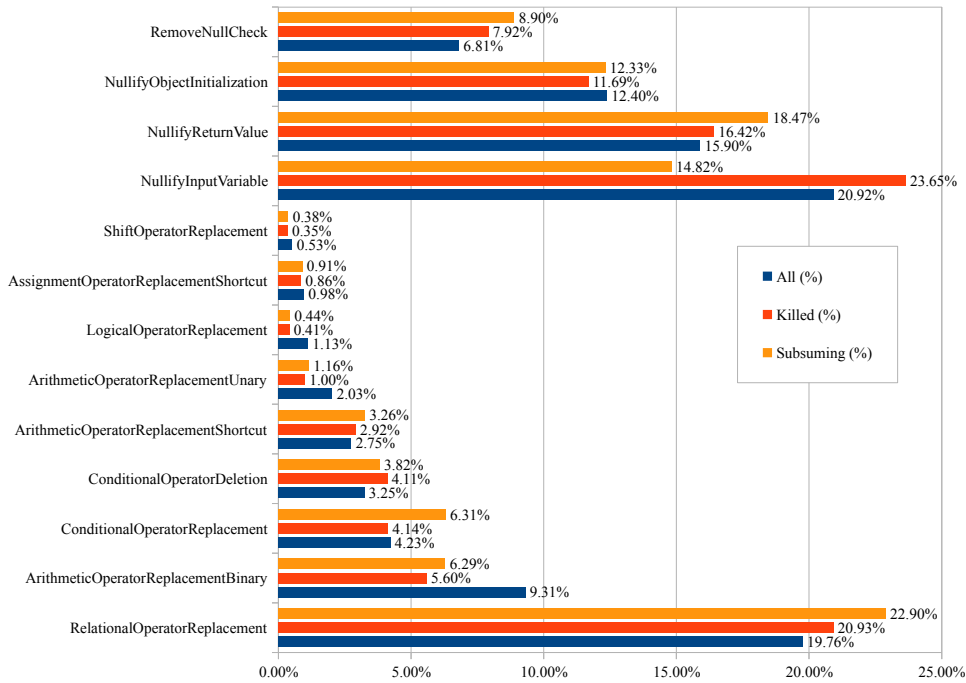


Figure 7.7: Ratio of Mutants by Each Mutation Operator in All, Killed, and Subsuming

7.5 THREATS TO VALIDITY

To describe the threats to validity we refer to the guidelines reported by Yin [105]. Threats to **internal validity** focus on confounding factors that can influence the obtained results. These threats stem from potential faults hidden inside our analysis tools. While theoretically possible, we consider this chance limited. The tools used in this experiment have been used previously in several other studies, and their results went through many iterations of manual validation. In addition, the code of LittleDarwin and LittleDarwin-Null along with all the raw data of the study is publicly available for download in the replication package [132].

Threats to **external validity** refer to the generalizability of the results. In RQ1 we advocate for the adoption of null-type mutation operators by using a didactic project. We alleviate the non-representativeness of this project, by analyzing 15 real open-source projects in RQ2. Although our results are based on projects with various levels of test adequacy in terms of traditional and null-type mutation coverage, we cannot assume that this sample is representative of all Java projects. We use PITest, LittleDarwin, and Javalanche as mutation testing tools. We cannot assume that these tools are representative

of all mutation tools available in literature. For this reason, we refer to these tools as general-purpose since they can work with little effort on many open-source projects. We modeled null-types mutation operators upon the typical null-type faults described by Osman et al. [122]. However, there may be other types of null-type faults that we did not consider. There are also other tools (such as PITest) that have similar mutation operators to our proposed ones. Even though such mutation operators may have existed before, our study is the first time that null-type mutation operators are evaluated extensively. Even if this was the case, our results should still hold since we already demonstrate with four mutation operators that they are in need of explicit modeling.

Threats to **construct validity** are concerned with how accurately the observations describe the phenomena of interest. The problem of equivalent mutants affects the analysis of surviving mutants on the test suites of the 15 open-source projects. Due to the large number of created mutants, it is impractical to filter equivalent mutants in the final results. Still, we believe this threat is minimal, because we analyze two different aspects of mutation testing, which lead to converging results. The total number of generated mutants can be different based on the set of mutation operators that are used in each tool. However, this difference has been taken into account when discussing the results of the experiments. To measure redundancy among the mutants, we use dynamic subsumption relationship. However, the accuracy of the dynamic subsumption relationship depends on the test suite itself. This is a compromise, as the only way to increase the accuracy is to have several tests that kill each mutant, which is not practical.

7.6 CONCLUSION

Developers are prone to introduce null-type faults in Java programs. Yet, there is no specific approach devoted to helping developers strengthen the test suite against these faults. On the one hand, mutation testing provides a systematic method to create tests able to prevent common faults. On the other hand, the general-purpose mutation testing tools available today do not model null-type faults explicitly by default.

In this paper, we advocate for the introduction of null-type mutation operators for preventing null-type faults. As a first step, we show that traditional mutation operators are not enough to cope with null-type faults as they cannot lead to the creation of a mutation-adequate test suite that can kill all of them. Then we demonstrate, by means of code examples, how the null-type mutants can drive the extension of the test suite. Finally, we highlight that null-type mutation operators are helpful in practice by showing on 15 open-source projects that real test suites are inadequate in detecting null-type faults. In this context, we explore the trade-offs of having null-type mutants. On the downside, we show that the inclusion of null-type mutants increases the mutant redundancy. Yet,

this increment is only marginal. On the upside, we show that null-type mutants complement traditional mutants in two ways. First, they provide a large number of survived mutants to the developer to strengthen the test suite. Second, they comprise a large part of subsuming mutants.

As a consequence, developers can increase their confidence in the test suite regarding to the null-type faults by (i) prioritizing the classes that have a large difference in traditional and null-type mutation coverage, (ii) creating tests to kill the survived null-type mutants in these classes, and (iii) repeating the process until all classes have similar levels of traditional and null-type mutation coverage.

C++11/14 Mutation Operators Based on Common Fault Patterns



C++11/14 Mutation Operators Based on Common Fault Patterns

Ali Parsai, Serge Demeyer, and Seph De Busser

In *Proceedings of 2018 International Conference on Testing Software and Systems (ICTSS 2018)*, 102–118. Cadiz, Spain. October, 2018.

URL: https://doi.org/10.1007/978-3-319-99927-2_9.

This chapter was originally published in the *Proceedings of 2018 International Conference on Testing Software and Systems (ICTSS 2018)*.

CONTEXT

This chapter targets the second of our three identified problems, the fault model problem. In particular, it aims to introduce new mutation operators that target the C++11/14 constructs in C++. These new constructs have been identified by the domain experts as a source of potential faults, and several fault patterns have been presented in gray literature in recent years. We introduce four mutation operators that allow developers to write tests targeting such faults.

We added more information about Range-Based For to this chapter compared to the originally published version.

ABSTRACT

The C++11/14 standard offers a wealth of features aimed at helping programmers write better code. Unfortunately, some of these features may cause subtle programming faults, likely to go unnoticed during code reviews. In this paper we propose four new mutation operators for C++11/14 based on common fault patterns, which allow to verify whether a unit test suite is capable of testing against such faults. We validate the relevance of the proposed mutation operators by performing a case study on seven real-life software systems.

8.1 INTRODUCTION

Nowadays, the process of software development relies more and more on automated software tests due to the developers interest in testing their software components early and often. The level of confidence in this process depends on the quality of the test suite. Therefore, measuring and improving the quality of the test suite has been an important subject in literature. Among many of the studied techniques, mutation testing is known to perform well for improving the quality of the test suite [81].

The idea of mutation testing is to help identify software faults indirectly by improving the quality of the test suite through injecting an artificial fault (i.e. generating a *mutant*) and executing the unit test suite to see whether the fault is detected [23]. If any of the tests fail, the mutant is said to be detected, thus *killed*. On the other hand, if all the tests pass, the test suite failed to detect the mutant, thus the mutant *survived*. However, some mutants result in code which does not pass the compiler and these are called *invalid mutants*. And in other situations, a mutant fails to change the output of a program for any given input hence can never be detected — these are called *equivalent mutants*.

A mutant is created by applying a transformation rule (i.e. *mutation operator*) to the code that results in a syntactic change of the program [31]. Given an effective set of mutation operators, mutation testing can help developers identify the weaknesses in the test suite [133]. Nevertheless, designing effective mutation operators requires considerable knowledge about the coding idioms and the common programming faults often made in the language [31]. More importantly, good mutation operators should maximize the likelihood of *valid* and *non-equivalent* mutants [134].

The first set of mutation operators were reported in King et al. [48]. They were later implemented in the tool Mothra which was designed to mutate the programming language FORTRAN77. With the advent of the object-oriented programming paradigm, new mutation operators were proposed to cope with specific programming faults therein [90]. This is a common trend in mutation testing: languages evolve to get new language constructs; some of these constructs cause subtle programming faults; after which new mutation operators get designed to shield against these common faults. For example, with

the evolution of Java related languages, mutation operators have been designed to account for concurrent code [124], aspect-oriented programming [125], graphical user interfaces [126], and Android applications [127].

The C++11/14 standard (created in 2011 and 2014 respectively) offers a wealth of features aimed at helping programmers write better code [135]. Most notably there is more type-safety and compile-time checking (e.g. `static_assert`, `override`). Unfortunately, the standard also provides a few features that may cause subtle faults (e.g. lambda expressions, list initialization, ...). Our goal is to identify these sources of common faults and introduce new mutation operators that address them. While it is possible that some subset of these faults are addressed by C++99 mutation operators, previous experience shows targeted mutation operators prove useful in improving the test suite quality further [17, 136].

In this study, we seek to answer the following research questions:

- **RQ1.** Which categories of C++11/14 faults are most likely to be made by programmers, and what are the corresponding mutation operators?
- **RQ2.** To what extent do these mutation operators create valid, non-equivalent mutants?

The rest of this paper is structured as follows: In Section 8.2 we provide the necessary background information about this study, and briefly discuss the related work. In Section 8.3 we discuss our approach to answering our research questions, and show our results in Section 8.4. Finally, we present our conclusions in Section 8.5 and highlight the future research directions rooted in this work.

8.2 BACKGROUND AND RELATED WORK

In this section we provide the necessary background information needed to comprehend the rest of the article and discuss the related work. First, we describe mutation testing and its related concepts. Then, we describe the new C++11/14 features, focusing on subtle faults that may be revealed via mutation testing.

8.2.1 Mutation Testing

Mutation testing is the process of inserting bugs into software (*Mutants*) using a set of rules (*Mutation Operators*) and then running the accompanying test suite for each inserted mutant. If all tests pass, the mutant survived. If at least one test fails, the mutant is killed. If the mutant causes an error during compilation of the production code, it is invalid. A valid mutant that does not change the semantics of the program, thus making it impossible to detect, is called equivalent.

An equivalent mutant is a mutant that does not change the semantics of the program, i.e. its output is the same as the original program for any possible input. Therefore, no test case can differentiate between an equivalent mutant and the original program, which makes it undesirable. The detection of equivalent mutants is undecidable due to the halting problem [42]. The only way to make sure there are no equivalent mutants in the mutant set is to manually inspect and remove all the equivalent mutants. However, this is impractical in practice. Therefore, the aim is to generate as few equivalent mutants as possible.

Mutation operators are the rules mutation testing tools use to inject syntactic changes into software. Most operators are defined as a transformation on a certain pattern found in the source code. The first set of mutation operators ever designed were reported in King et al. [48]. These mutation operators work on basic syntactic entities of the programming language such as arithmetic, logical, and relational operators. Offutt et al. came up with a selection of few mutation operators that are enough to produce high quality test suites with a four-fold reduction of the number of mutants [49]. Kim et al. extended the set of mutation operators for object-oriented programming constructs [90].

Because of the complexity of parsing C++, building a mutation testing tool for C++ is almost equivalent to building a complete compiler [137]. It is only with modern tooling, e.g. the Clang/LLVM compiler platform, that it became possible to write such tools without an internal parser.

Kusano et al. developed CCmutator, a mutation tool for multi-threaded C/C++ programs that mutates usages of POSIX threads and the C++11 concurrency constructs, but works on LLVM's intermediate representation instead of directly on C++ source code [138]. Delgado-Perez et al. have expanded on the work done for the C language by adding class mutation operators, and created a set of C++ mutation operators [136]. In addition, they show that the class mutation operators compliment the traditional ones and help testers in developing better test suites.

8.2.2 C++11/14

C++11 was introduced in 2011 with the goal of adapting C++ and its core libraries to modern use cases of the language (e.g. multi-threading, genetic algorithms, ...). This release was followed by C++14 in 2014 with similar goals. The introduction of C++11/14 has changed the language to the point that earlier iterations of the language are dubbed the classical C++, and modern C++¹ starts with C++11/14. The release of the standard was followed by real-time adoption in compilers such as Clang and G++.

Unfortunately, the C++11/14 standard also provides a few features that may cause

¹<http://www.modernesccpp.com/index.php/what-is-modern-c>

subtle faults, thus where support in the form of new mutation operators would be desirable. In this subsection we briefly explain these features of C++11/14.

Range-Based For Loop

Range-Based For Loop [<http://en.cppreference.com/w/cpp/language/range-for>] is made to simplify looping over a range of elements, and generalizes the use of for loop to any container with a ForwardIterator and a begin() and end() method. For example, the following two loops give similar results despite the fact that in the loop on the left *i* represents an element in the vector, while the *i* in the right loop is a subscript to the vector:

```
for (int i : v) {
    std::cout << i << '\n'; }
```

```
for (int i=0; i<v.size(); i++) {
    std::cout << v.at(i) << '\n'; }
```

Lambda Expressions

Lambda Expressions [<http://en.cppreference.com/w/cpp/language/lambda>] allow for the definition of unnamed in-line functions. For example, in the following piece of code, lambda contains a function which *captures* *a* and *b* (they are available in the body of lambda as const expressions), takes an input parameter *x*, and returns a bool.

```
int a, b;
auto lambda = [a, b](int x) {return x > a + b;}
```

It is possible to have a default capture at the start of the capture list, e.g. '=' for by-value, or '&' for by-reference capture. This causes all variables referenced in the lambda body to be captured the specified way.

Move Semantics

Move Semantics [http://en.cppreference.com/w/cpp/language/move_constructor] are introduced in C++11/14 to address the inefficiencies of copy construction when the copied value is deleted after the execution of the constructor. For example, the following code would be inefficient in C++03:

```
std::vector<int> v(ComputeLargeVector(1000));
```

In C++03, this code would create the vector in `ComputeLargeVector`, call the copy constructor for `v`, which copies all elements into a newly allocated buffer, and then destroys

the original. With move semantics, `v` would simply copy the internal size, capacity, and pointer to the elements in the temporary vector and set the members of the temporary vector to 0.

To enable this, value categories² got redefined in C++11. Every expression is either an lvalue, an xvalue, or a prvalue. The difference between these value categories lies in two properties: whether or not they have identity (i.e. it is possible to determine whether two expressions are the same using an address), and whether they can be moved from (move semantics can bind to the expression). lvalues and xvalues have identity, while xvalues and prvalues can be moved from. All rvalues can bind to rvalue references, which are denoted by `&&`. For example, the signature of the move constructor of vector is:

```
vector<T> (vector<T>&&);
```

It is possible to convert an lvalue to an xvalue through `std::move`, which casts the object to an rvalue reference type.

Perfect Forwarding

Perfect Forwarding [<http://en.cppreference.com/w/cpp/utility/forward>] allow for forwarding of input arguments to other functions as-is. For example, the `emplace` family of functions in the standard containers accept any number of arguments and forward them to the constructor of the element type. The following template function constructs an object of type `T` with a given argument:

```
template<typename T, typename Arg>
T construct(Arg&& argument) {
    return T{std::forward<Arg>(argument)};
}
```

Because `Arg` is a template parameter, `Arg&&` is a forwarding reference [139]. This means that it will resolve to either an lvalue or an rvalue reference depending on argument. If `argument` is an lvalue, `std::forward` is a no-op, and if `argument` is an rvalue reference, it behaves the same way `std::move` does.

List Initialization

List Initialization [http://en.cppreference.com/w/cpp/language/list_initialization] is a new syntax introduced in C++11 that allows the initialization of an object from braced initial values. It expands the ability to construct structs and arrays using braced initial-

²http://en.cppreference.com/w/cpp/language/value_category

izer to all types in C++. For example, the following is a valid syntax for creating and initializing an array of *int*:

```
int b {1,2,3,4,5};
```

Also, a type with a constructor that takes `std::initializer_list` as an argument can be initialized using this new syntax. For example, the following declaration of a `std::vector` creates a vector of integers with 5 elements:

```
std::vector<int> v{1,2,3,4,5};
```

8.3 STUDY DESIGN

In this section, we discuss the design of our study. First, we explain our evaluation criteria, and then we describe the process by which we determine the fault categories and create mutation operators. Finally, we present the details of our data set.

8.3.1 Evaluation Criteria

RQ1. Which categories of C++ 11/14 faults are most likely to be made by programmers, and what are the corresponding mutation operators?

To evaluate the results of this question, the mutation operator needs to fulfill the following criteria:

- Can the mutation operator simulate a fault from the fault category we identified?
- Is it reasonable to assume that the software developer can create faulty code similar to the generated fault?

We look at guidelines provided by experts concerning the new standards and the common pitfalls mentioned therein. We search for such patterns and select those that can be reconstructed into a mutation operator.

RQ2. To what extent do these mutation operators create valid, non-equivalent mutants?

$$\text{Mutation Operator Score} = 1 - \frac{E - D}{T - I - D} \quad (8.1)$$

T = Total Number of Mutants, E = Number of Equivalent Mutants, D = Number of Easily-Detectable Equivalent Mutants, I = Number of Invalid Mutants

An effective mutation operator generates valid semantic faults. This means that mutation operators need to generate as few equivalent mutants as possible. We borrow this

criterion from Delgado-Perez et al. who used it in their study [134]. It is also important for each mutant to be valid, i.e. the mutated program compiles without errors. Any mutation operator inevitably generates equivalent mutants. Some of these equivalent mutants, however, can be identified and filtered at compile-time. We call these equivalent mutants as easily-detectable equivalent mutants. To quantify the effectiveness of each mutation operator, we calculate the percentage of equivalent mutants among the valid mutants after filtering the easily-detectable equivalent mutants. The mutation operator score is then calculated by deducting the mentioned percentage from 100% (see Equation 8.1). For each mutation operator, we provide methods to filter easily-detectable equivalent mutants.

To see how our operators work in real-life scenarios, we looked at seven open source projects that are using C++11/14 (see Table 8.1). Our analysis consists of applying our mutation operators to create all possible mutants. We do this by manually searching for the code patterns that match (using `grep`). Then, we manually categorize the resulting mutants into invalid, equivalent, and valid non-equivalent mutants. If a mutant did not change the semantics of the program, we classified it as an equivalent mutant. If the operator created a non-compilable program, we classified the mutant as invalid. Otherwise, we considered the mutant as valid non-equivalent.

8.3.2 Data Set

In this subsection, we present the details of our data set. Our data set is publicly available in the replication package available at <https://www.parsai.net/files/research/ICTSSRepPak.zip>.

In order to find the common fault patterns related to C++11/14, we looked at the authoritative sources of fault patterns such as those suggested by Scott Meyers in his book titled *Effective Modern C++* [140], and *C++ Core Guidelines* by Bjarne Stroustrup [141]. We also took into account the standard proposal N3853 by Stephan Lavavej [142] which points out problems with range-based for loop syntax.

For the evaluation of the mutation operators, we looked at seven open source projects that use C++11/14 (Table 8.1). These projects range from a small, several hundred lines of code header-only library, to a full application with over 100,000 lines of code with years of active development:

- *i-score*³ is an interactive intermedia sequencer, built in Qt.
- *C++React*⁴ is a C++11 reactive programming library, based on signals and event

³<https://github.com/OSSIA/i-score/>

⁴<https://github.com/schlangster/cpp.react>

Table 8.1: Project Statistics

Project	Commit	Size(Lines of Code)		Number of Commits	Team Size
		Production	Test		
i-score	c86cd3d	108K	3.5K	5358	14
C++React	1f6ddb7	11K	2K	417	1
EntityX	6389b1f	9K	1K	296	28
Antonie	59deb0d	9K	0.1K	306	2
Json	a09193e	8K	18K	1973	59
Corrade	ff3b351	6.5K	9.1K	1898	10
termdb	bd0fb4a	783	153	26	2

streams.

- *EntityX*⁵ is an Entity Component System that uses C++11 features.
- *Antonie*⁶ is a processor of DNA reads, developed at the Bertus Beaumontlab of the Bionanoscience Department of Delft University of Technology.
- *Json*⁷ is a single-header library for working with Json with modern C++.
- *Corrade*⁸ is a C++11/14 utility library, including several container classes, a signal-slot connection library, a unit test framework, a plugin management library and a collection of other small utilities.
- *termdb*⁹ is a small C++11 library for parsing command-line arguments.

8.4 RESULTS

In this section, we present the results of our research. For each mutation operator, first we give its definition, then we discuss the motivation behind it to answer RQ1, and finally we provide our analysis of the data set to answer RQ2.

8.4.1 FOR

The range-based “for” reference removal (FOR) operator finds instances of range-based for loops of the form `for (T& elem : range)` or `for (T&& elem : range)`, where T is either auto or a concrete type, and removes the reference qualifier from the range declaration. Table 8.2 shows the results for this mutation operator.

⁵<https://github.com/alecthomas/entityx>

⁶<https://github.com/beaumontlab/antonie>

⁷<https://github.com/nlohmann/json>

⁸<https://github.com/mosra/corrade>

⁹<https://github.com/agaunoyal/termdb>

Code Excerpt 8.1: Original For

```
for(auto& elem : range) { ... }
```

Code Excerpt 8.2: Mutated For

```
for(auto elem : range) { ... }
```

Motivation (RQ1).

FOR operator is based on the possibility of confusion over the default value semantics of the new range-based for loop, whereas previous methods of looping over containers resulted in reference semantics. This was noted previously by Stephan Lavavej [142]. In his standard proposal, he lists three problems with the most idiomatic-looking range-based for loop, `for (auto elem : range)`, namely:

- It might not compile - for example, `unique_ptr`¹⁰ elements are not copyable. This is problematic both for users who won't understand the resulting compiler errors, and for users writing generic code that'll happily compile until someone instantiates it for movable-only elements.
- It might misbehave at runtime - for example, `elem = val;` will modify the copy, but not the original element in the range. Additionally, `&elem` will be invalidated after each iteration.
- It might be inefficient - for example, unnecessarily copying `std::string`.

From a mutation testing perspective, the second reason is the main motivation to create a mutation operator. In the case of a range-based for loop that modifies the elements of a container in-place, the correct and generic way to write it is `for (auto&& elem : range)`. For all cases except for proxy objects and move-only ranges, `for (auto& elem : range)` works as well.

This operator is only a minor syntactic change that is easily overlooked even in code review if such fault pattern is not actively looked for. Surviving mutants of this type can pinpoint the loops whose side effects on container elements are not tested.

Analysis (RQ2).

Invalid Mutants: The invalid mutants are comprised of two groups. The majority of the invalid loops were over containers of move-only types. Of the invalid mutants in *i-score*, 33 were containers of pointers to virtual interface classes with custom dereferencing iterators, making the mutant try to instantiate a non-instantiable type. Both of these cases can be easily checked when generating the mutants.

¹⁰http://en.cppreference.com/w/cpp/memory/unique_ptr

Table 8.2: Results of FOR Operator

Project	Total	Invalid	Equivalent	Easily Detectable	Score
i-score	251	101	115	110	87.5%
Corrade	24	1	13	13	100%
Json	1	0	0	0	100%
EntityX	2	0	2	2	N/A
termdb	0	0	0	0	N/A
C++React	8	0	6	6	100%
Antonie	39	10	18	18	100%

Equivalent Mutants: In the majority of equivalent cases, the body of the loop did not mutate the referenced element in the container, thus making it equivalent to a loop with an added `const` qualifier. This is relatively easy to verify automatically, hence such mutants are listed as detectable. Only a handful of equivalent cases were loops that did mutate the elements of the container, but the container never gets used after the loop finishes. This would require more complicated static analysis.

8.4.2 LMB

The lambda reference capture (LMB) operator changes a default *value* capture to a default *reference* capture. Table 8.3 shows the results for this mutation operator.

Code Excerpt 8.3: Original Lambda

```
[=](int x) { return x + a; };
```

Code Excerpt 8.4: Mutated Lambda

```
[&](int x) { return x + a; };
```

Motivation (RQ1).

This operator is based on the warnings on default capture modes in Core Guideline F53 and Meyers' 31st item [140, 141]. This mutation operator results in code that leads to undefined behavior if the lambda is executed in a non-local context, because the references to local variables are not valid. This can happen when the lambda is pushed up the call stack or sent to a different thread for asynchronous execution.

Just like the FOR operator, this operator is only a minor syntactical change that can easily be overlooked, and results in faults that are not necessarily easy to detect; thus it is worth testing for its absence. Mutants created by this operator are not easy to detect either, because they invoke undefined behavior which is highly dependent on compiler

Table 8.3: Results of LMB Operator

Project	Total	Invalid	Equivalent	Easily Detectable	Score
i-score	189	0	113	101	86.3%
Corrade	0	0	0	0	N/A
Json	0	0	0	0	N/A
EntityX	0	0	0	0	N/A
termdb	0	0	0	0	N/A
C++React	1	0	0	0	100%
Antonie	0	0	0	0	N/A

optimization levels and runtime circumstances.

Analysis (RQ2).

Invalid Mutants: We did not witness any invalid mutants generated by this operator in our data set.

Equivalent Mutants: All undetectable equivalent mutations were ones where the lambda gets passed into a function that executes it within its own scope. While it is theoretically possible to detect them, we classify them as undetectable because it would require complicated non-local reasoning. The other equivalent mutants are detectable by taking into account what the capture list actually captures. For example, in Code Excerpt 8.5, the minimal capture list is empty, whereas in Code Excerpt 8.6 the minimal capture list is [a] and in Code Excerpt 8.7 the minimal capture list is [this]. In the first and third examples, replacing the default value-capture with reference-capture changes nothing about the capture list. In i-score, these made up the majority of equivalent cases, hence the high percentage of detectable equivalent mutants.

Code Excerpt 8.5: Empty Capture

```
[=](int x) {return x < 1;};
```

Code Excerpt 8.6: Local Capture

```
int a; [=](int x) {return x < a;};
```

Code Excerpt 8.7: 'this' Capture

```
struct Foo {
  int a;
  auto getFilter() {
    return [=](int x) {return x < a;};
  }
};
```

8.4.3 FWD

The forced rvalue forwarding (FWD) operator replaces `std::forward` instances with `std::move` to force moving from forwarded arguments. Table 8.4 shows the results for this mutation operator.

Code Excerpt 8.8: Original Forwarding

```
template<class T>
void wrapper(T&& arg)
{
    foo(std::forward<T>(arg));
}
```

Code Excerpt 8.9: Mutated Forwarding

```
template<class T>
void wrapper(T&& arg)
{
    foo(std::move(arg));
}
```

Motivation (RQ1).

There are often two possible errors in relation to forwarding semantics (which Meyers warns about in his items 24 and 25 [140]): forgetting to use `std::forward` (and thus passing both lvalues and rvalues on as lvalues) or moving instead of forwarding (and thus passing lvalues on as rvalues to be moved from).

As an example, the following function constructs an object of type `T` using uniform initialization by forwarding the variadic list of arguments using perfect forwarding:

```
template<typename T, typename... Args>
T construct(Args&&... args) {
    return T{std::forward<Args>(args)...};
}
```

We then use the following type, chosen because `std::string` has a destructive move constructor and `std::unique_ptr` is a move-only type:

```
struct Widget
{
    std::string text;
    std::unique_ptr<int> value;
};
```

Then the following code constructs two `Widgets` with the same text and different values:

```
std::string text{64,'a'}; //Long enough to disable SSO
auto w1 = construct<Widget>(text, std::make_unique<int>(0));
auto w2 = construct<Widget>(text, std::make_unique<int>(1));
```

Table 8.4: Results of FWD Operator

Project	Total	Invalid	Equivalent	Easily Detectable	Score
i-score	71	13	18	9	81.6%
Corrade	5	0	0	0	100%
Json	14	0	14	6	0%
EntityX	7	0	1	1	100%
termdb	0	0	0	0	N/A
C++React	160	0	17	15	98.6%
Antonie	0	0	0	0	N/A

Both calls result in `Args` being `[std::string&,std::unique_ptr<int>&&]`, which makes `std::forward` correctly forward the first argument as `lvalue` and the second as `rvalue`. Forgetting to use `std::forward` results in both arguments being forwarded as `lvalues`, which fails to compile since `std::unique_ptr` is a move-only type. When forgetting to forward, code will always either compile and default to copying the types, or fail to compile because a move-only type is used. Since for all types, the only visible effect of doing a copy instead of a move is a performance degradation, this would not be a useful operator for testing purposes.

Replacing the `std::forward` with `std::move`, however, does have the potential to change program behavior. With `construct` mutated as in the code sample above, the string `text` will be moved from in the first call, and the second call results in unspecified behavior. In most standard library implementations, `w2` will end up with an empty `text`. Meyers argues that it is easy to confuse `rvalue` and forwarding references because of their identical syntax, making this a likely fault for developers to make.

A large part of these mutants can be targeted by using forwarding on a non-const `lvalue` argument, since it cannot bind to an `rvalue` reference. Another way of testing these is to use a type with a destructive move, and test the state of the original object after passing it into the function as an `lvalue`.

Analysis (RQ2).

Invalid Mutants: The invalid mutants were comprised of two groups: fixed template argument and non-const `lvalue` reference callee arguments. The first group forwards to another template function while explicitly stating the template argument as seen in Code Excerpt 8.10. This causes the code to not compile when called with a non-const `lvalue`. If it is called with const `lvalues` or `rvalue` references it will have the same runtime behavior as the original.

Code Excerpt 8.10: Fixed Template Argument Forwarding

```

template<typename T>
void foo(T&&);

template<typename T>
void bar(T&& t) {
    foo<T>(std::forward<T>(t));
}

```

The second group forwards into a function with fixed arguments, at least one of which is a non-const lvalue reference, as seen in Code Excerpt 8.11 which defines a function that calls another with a prepended integer argument. Because the second argument is a non-const lvalue reference, applying the operator here results in an invalid mutant because it cannot bind to an rvalue reference.

Code Excerpt 8.11: Forwarding into Non-Const Lvalue Reference

```

void foo(int, int&, int);

template<typename ... Args>
void bar(Args&&... args) {
    foo(1, std::forward<Args>(args)...);
}

```

Equivalent Mutants: There are three categories of equivalence for this operator. The first is where `std::forward` gets used within a `decltype` or `noexcept` context, where the operator either changes nothing, or makes the code fail to compile. This is why we classify these as detectable equivalent mutants. The second case is where the forwarded argument never gets stored, which makes irrelevant the difference between `std::forward`, `std::move`, and passing by reference. The third and final category is where the callees are guaranteed to not take rvalue references or value parameters of movable types. Of these three categories, the first is easily detectable by filtering out mutants within a `decltype` or `noexcept` expression. The second would require sophisticated flow analysis which is why we listed them as not easily-detectable. The last category can be detected if it is feasible to find all possible callees and see whether they take any rvalue references or value parameters of movable types. This is only feasible for mutants calling functions that cannot be overloaded by external code, since it is otherwise theoretically possible to introduce a new overload of the called function that takes a parameter of a type with a destructive move, making the mutant non-equivalent. The mutants for which this analysis is possible are listed as detectable in our analysis.

Table 8.5: Results of INI Operator

Project	Total	Invalid	Equivalent	Easily Detectable	Score
i-score	1	0	0	0	100%
Corrade	0	0	0	0	N/A
Json	0	0	0	0	N/A
EntityX	0	0	0	0	N/A
termdb	1	0	0	0	100%
C++React	0	0	0	0	N/A
Antonie	18	0	0	0	100%

8.4.4 INI

The initializer list constructor (INI) operator checks constructor calls of types with an initializer list constructor and changes to/from uniform initialization in order to provoke calling a different constructor. Table 8.5 shows the results for this mutation operator.

Code Excerpt 8.12: Original Initializer

```
std::vector<int> v(3,2);
```

Code Excerpt 8.13: Mutated Initializer

```
std::vector<int> v{3,2};
```

Motivation (RQ1).

While initializer list constructors are helpful in defining container contents, they are possible sources of faults as well. For example, when using uniform initialization one needs to pay attention to the correct syntax, since using {} instead of () by mistake changes the semantics of the expression drastically. A prominent example of this problem is `std::vector` of integer types, which Meyers points out in his 7th item [140]. The non-mutated version in Code Excerpt 8.12 defines a vector of three elements with value 2, whereas the mutated vector in Code Excerpt 8.13 has only two elements: 3 and 2.

Analysis (RQ2).

Invalid Mutants: This operator has no way of creating invalid mutants by design, because it checks whether or not a different constructor is called when it is applied. This includes checking for narrowing conversions; e.g. when trying to mutate `std::vector<char>(10, 'a');`.

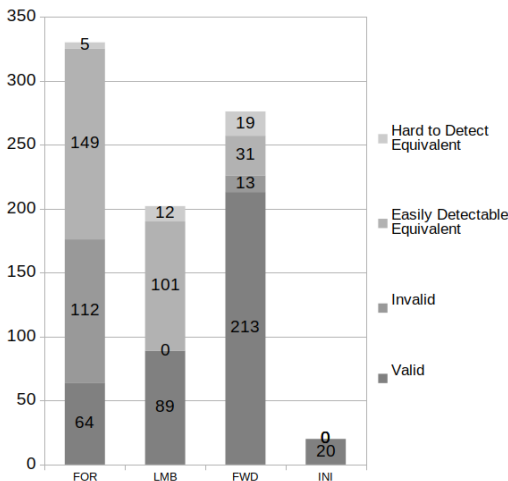


Figure 8.1: Generated Mutants

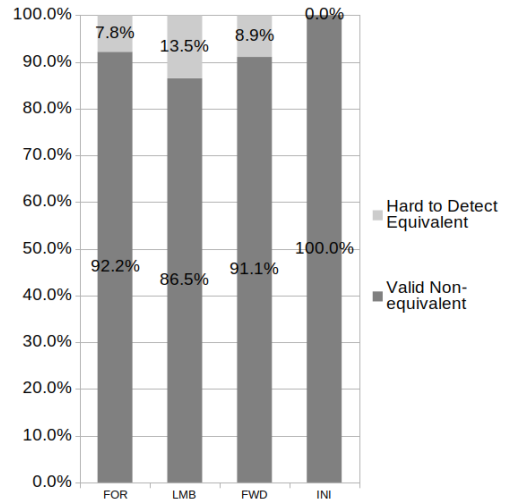


Figure 8.2: Mutation Operator Scores

Code Excerpt 8.14: Equivalence Cases for INI

```

struct Default1 {
int foo = 1;
Default() = default;
Default(int f) : foo(f) {};
};

std::vector<Default1> v1(1); //v1{1}
std::vector<int> v2(2,2); //v2{2,2}

```

Equivalent Mutants: There are only a few corner cases for `std::vector` where this operator results in equivalence (e.g. Code Excerpt 8.14).

In both of these cases, the mutated initializer results in the same vector as the original. Given the number of times this pattern was observed in our data set (20 instances in all projects), it is unlikely that such equivalent mutants are found in any significant number.

8.4.5 Discussion

We have aggregated the number of all generated mutants per kind for each mutation operator in Figure 8.1. The FOR operator generates the highest number of mutants, most of which are either invalid or easily detectable equivalent. Hence, it is possible to filter most of these mutants easily. This is why this mutation operator is promising. The most promising mutation operator is INI, which generated no invalid or equivalent mutants in our data set. However, the low number of mutants generated by this mutation operator

means that it might not be applicable in every case. FWD is the operator that generates the most valid, non-equivalent mutants along with a low number of equivalent and invalid mutants, while LMB generates no invalid mutants at all but has a slightly higher ratio of equivalent mutants that are hard to detect.

Figure 8.2 shows the mutation operator score for each mutation operator. It is clear that all mutation operators are within reasonable boundaries regarding the percentage of generated hard to detect equivalent mutants when compared to other C++ mutation operators (e.g. Delgado-Perez et al. [134]). Overall, we found that these mutation operators have a high mutation operator score, with all of them generating very few equivalent mutants (13.5% or less of the total number of mutants).

One of the noticeable trends among these mutation operators is their tendency to generate lots of mutants in a single project, and few in others. For example, INI generated 18 mutants in *Antonie*, and 2 in all other projects, while LMB generated 189 mutants in *i-score* and only 1 in others. Other than the size of the projects, we found that the adoption of the new syntax has not been uniform in all of the projects, i.e. some projects make use of mostly a single new syntactic feature and not all of them.

8.5 CONCLUSIONS AND FUTURE WORK

In this study, we created a set of mutation operators that target the common faults introduced by C++11/14 syntactic features. We collected advice about the new C++11/14 syntax from authoritative sources, and created four new statement-level mutation operators (FOR, LMB, FWD, and INI). For each mutation operator, we discussed the motivation behind its creation and the type of faults they generate. We used Mutation Operator Score as a way to measure the effectiveness of each mutation operator. For this, we selected 7 real-life C++11/14 projects, and counted the number of valid, invalid, easily detectable and hard to detect equivalent mutants generated by each mutation operator for each project. Our results show that all of the introduced mutation operators generate at most 13.5% hard to detect equivalent mutants. The high operator scores indicate that these mutation operators are a useful addition to the mutation operators suggested previously in literature.

Several aspects of this study can be researched further. In particular, the use of our proposed mutation operators alongside traditional and class mutation operators may result in finding multiple redundancies among these mutation operators. In addition, a comparative study similar to Delgado-Perez et al. [136] between these mutation operator sets would provide more insight into the usefulness of each set of operators depending on the context.

Comparing Mutation Coverage Against Branch Coverage in an Industrial Setting



Comparing Mutation Coverage Against Branch Coverage in an Industrial Setting (Under Review)

Ali Parsai and Serge Demeyer

In *International Journal on Software Tools for Technology Transfer (Under Review) (STTT)*, .. August, 2019.

URL: N/A.

This chapter is submitted to *International Journal on Software Tools for Technology Transfer (Under Review) (STTT)*.

CONTEXT

This chapter targets the third of our three identified problems, the tool problem. In particular, it aims to compare the test quality metric preferred in practice (branch coverage) to theory (mutation coverage). In this chapter, we demonstrated the feasibility of performing mutation testing on complicated industrial software. By adapting LittleDarwin to be used in the continuous integration environment of an industrial company and performing a case study, we showed that it is possible to adapt and use mutation testing in practice, and indeed acquire valuable information about the test suite.

ABSTRACT

The state of the practice in software development is driven by continuous integration: frequent and fully automated tests in order to detect faults immediately upon project build. As the fault detection capability of the test suite becomes so important, modern software development teams continuously monitor the quality of the test suite as well. However, it appears that the state of the practice is reluctant to adopt strong coverage metrics (namely mutation coverage), instead relying on weaker kinds of coverage (namely branch coverage). In this paper, we investigate three reasons that prohibit the adoption of mutation coverage in a continuous integration setting: (1) the difficulty of its integration into the build system, (2) the perception that branch coverage is “good enough”, and (3) the performance overhead during the build. Our investigation is based on a case study involving four open source systems and one industrial system. We demonstrate that mutation coverage reveals additional weaknesses in the test suite compared to branch coverage, and that it is able to do so with an acceptable performance overhead during project build.

9.1 INTRODUCTION

In software testing, developers assess the quality of test suite according to its capability of detecting yet unknown faults. For a faulty statement to be revealed, the program must pass through four stages: (i) the faulty statement must be executed (*Reachability*), (ii) the faulty statement needs to affect the program state (*Infection*), (iii) the effect of the faulty statement on program state need to propagate to the program output (*Propagation*), and the test needs to observe the failure in the program output (*Reveal*) [143, 144]. Several coverage metrics exist that aim to quantify the quality of a test suite. Among them, mutation coverage is generally acknowledged as the state-of-the-art coverage metric since it checks whether a test covers all four aforementioned stages [23, 24, 120, 145, 146]. Indeed, mutation testing is the process of deliberately injecting faults into a software system, and then verifying whether the tests actually fail. This faulty version of the software is called a *mutant*. The faults injected by each mutant are modeled after the common mistakes often made by developers, hence mutation testing provides a repeatable and scientific approach to measure the fault detection capability of a test suite [23, 31]. Using the results of mutation testing, it is possible to improve test suite quality or prioritize tests [147, 148, 149, 150]. Comparative studies demonstrated that in terms of fault detection, mutation testing is more effective than several other coverage criteria [24, 26, 151]. If the fault model used for mutation operators is close to reality, mutation testing produces more accurate results than simple coverage metrics [120]. Finally, mutation testing has been shown to subsume statement and branch coverage [152]. This is due to the fact that branch and statement coverage only check for reachability [145].

Despite these promising results, the state-of-the-art is not yet adopted into the state-of-

the-practice. For instance, Gopinath et al. put forward that mutation testing “*is generally not used by real-world developers with any frequency*” [29]. Instead, the state-of-the-practice relies on simple coverage metrics such as statement and branch coverage. This is despite their inadequacy for assessing the fault detection capability of a test suite [15, 16]. Among these metrics, branch coverage is commonly used in industry [12]. However, even with 100% of branch coverage there is still the potential for faults to go unnoticed, because branch coverage only checks whether the test code executes each branch and does not assess whether the program state is infected or the fault is propagated to the output [29, 153, 154]. Because of this, such coverage metrics cannot reveal anything regarding the quality of the test oracle.

A possible explanation for the preference towards simple coverage metrics is they are relatively easy and fast to collect. Today, there are plenty of test coverage tools available that instrument the code, execute the tests, and report the parts of the code not covered by the tests (i.e., statements, branches, paths). Also, while the performance overhead of these tools on the overall test execution are not negligible, it can often be tolerated in development environments [155, 156]. Moreover, Gligoric et al. demonstrate that branch coverage—among several coverage criteria—is the best one to predict the mutation coverage of a test suite [74]. From a practitioner’s point of view, branch coverage is a reasonable quality criterion because of the trade-off between time (the performance overhead induced by collecting the measurements) and quality (a “good enough” fault detection capability). This means that the successful adoption of mutation testing in industry requires convincing practitioners that there are tangible and worthwhile differences between the two methods when it comes to analyzing industrial software.

Literature blames the lack of adoption of mutation testing in industry mainly on the performance overhead, leading to the adagium —*do fewer, do smarter, and do faster* [30]. Examples in that sense are mutant sampling, weak mutation testing, and performing the mutation testing on byte code rather than the source code [23, 31]. However, the performance overhead is only one part of the equation. We argue that there are two other issues when adopting mutation testing in an industrial strength continuous integration setting: (i) the complexity of continuous build environments and (ii) the optimism regarding the added value of mutation testing. We expand on each of those issues below.

- (i) First, the research on mutation testing has ignored the novel trend that modern test infrastructure is part of a continuous integration environment. In such a setting, the build steps follow one another in lockstep. In fact, fully automatic test infrastructure is now common place in many companies, e.g. Google performs roughly 800 thousand builds and 150 million test runs in an average day automatically [157]. Integrating additional steps (such as calculating mutation coverage) easily interferes with such automated systems, and requires up-front considerations in the design

to ease the integration.

- (ii) Second, there is little empirical evidence that mutation testing reveals additional weaknesses in industrial strength test suites. Typically, studies that promote mutation testing are based on open source cases with components designed to be open and accessible [25, 26, 74, 76]. The same is not true for the industrial systems, where brown field development is common and legacy code is integrated as black-box components and tested accordingly [158, 159].

We derived these issues from a pilot study, integrating mutation testing in an industrial strength continuous integration setting. In particular, we were asked to explore the advantages and drawbacks of mutation testing in the context of the Segmentation component of the Impax ES medical imaging software used by Agfa HealthCare. The Segmentation component provides imaging algorithms to perform segmentation on 3D modeled volumes. As can be expected from a software system in healthcare, it must adhere to strict safety standards including advanced monitoring of test quality. Yet, the segmentation component interfaced with some legacy code, hence black-box testing was an inherent part of the testing strategy. Also, the Impax ES system is implemented as a small product-line, hence the Maven build system was configured to resolve dependencies with libraries depending on the product line variant to be built.

In our work, we first explore the feasibility of mutation testing in an industrial project which relies on a continuous integration system (RQ1). Then, we investigate the pros and cons that arise by its adoption. We attempt to verify that compared to branch coverage, mutation testing has the pros of revealing additional weaknesses in the test suite (RQ2), and it has the cons of introducing overhead (RQ3). This leads us to pursue the following research questions:

RQ1: *Is it feasible to integrate mutation testing in a continuous integration system?*

- ⇒ To answer this question we follow a proof by construction. We first integrate an existing mutation coverage tool (namely PITest) into the build system (namely Maven) We report the challenges we encountered and the workarounds we performed, all to no avail. Consequently, we adapted and used a mutation testing tool named *LittleDarwin* specifically designed to integrate well within a continuous integration environment.

RQ2: *Does mutation testing reveal additional weaknesses in the test suite compared to branch coverage?*

- ⇒ To answer this question we analyze branch coverage along with the mutation coverage. We investigate parts of the code where these two metrics differ to see whether mutation coverage indeed exposes additional weaknesses. To increase the generalizability of our findings, we perform the same comparison on four open source

systems.

RQ3: *Can we reduce the performance overhead induced by mutation testing to an acceptable level?*

⇒ To answer this question, we measure the performance overhead induced by a full mutation analysis, injecting 12K mutants for 38K lines of code. We compare this against the performance overhead after *mutant sampling* (Section 9.2.3), effectively reducing the number of mutants to 34.7%. We verify the results of the full mutation coverage against the sampled mutation coverage to see whether the reduced number of mutants still reveals the weaknesses in the test suite.

The rest of the paper is structured as follows. In Section 9.2, we provide some background information on test coverage in general and mutation testing in particular. We describe the main tools used in this study in Section 9.3. We then proceed with a discussion of the case study design, including a description of the cases under investigation as well as a detailed explanation of the setup for the open source and industrial cases in Section 9.4. The results of our case study are then discussed in Section 9.5, followed by a discussion of the threats to the validity of this study in Section 9.6. An overview of related work is presented in Section 9.7. Finally, in Section 9.8, we present our final conclusions.

9.2 BACKGROUND

In this section we present an overview of the background information necessary to understand the rest of the paper. We briefly introduce continuous integration environments and discuss the importance of test suite quality therein (Section 9.2.1), give a brief explanation of code coverage in general and branch coverage in particular (Section 9.2.2), and discuss mutation testing and its related concepts in detail (Section 9.2.3).

9.2.1 Testing in Continuous Integration Environment

Continuous integration is defined as the practice of merging the developed code with a central source code repository as often as possible. The concept of continuous integration was first proposed by Booch as a way to avoid integration problems [160]. This method has been in the center of attention in the past decade since it is the basis for today’s agile development techniques. Continuous integration allows the software to be continuously tested. Simple tasks (e.g. unit tests) can be triggered upon each commit; whereas, the time-consuming tasks (e.g. integration test) can be postponed to the nightly build. Providing continuous feedback, the continuous integration environment takes care that the code-base remains stable during development, and reduces the risk of arriving in integration hell (the point in production when members on a delivery team integrate their individual code) [161].

The introduction of agile development techniques has resulted in an increased interest in the fault detection capability of the test suite. This is typically monitored by means of *code coverage metrics*. In this context, a weakness in a test suite is a lack of capacity of a test suite to detect faults in a particular part of software, either by a lack of tests to cover that part, or incomplete testing of the covered part.

9.2.2 Code Coverage Metrics

Code coverage is defined as the proportion of code that is tested by the test suite. There are several ways to calculate code coverage. The most often used metrics in industry are statement coverage and branch coverage [29]. Statement coverage is the number of statements in the program that are executed at least once by the test suite divided by the total number of statements. Similarly, branch coverage is the number of branches executed at least once by the test suite divided by the total number of branches (Equation 9.1). Branch coverage subsumes statement coverage, because if all branches are examined, all statements contained in the branches are examined in the process [45, 162]. Branch coverage is often used in popular industrial tools to evaluate the quality of a test suite. More specifically, a high value of branch coverage is assumed to imply a “good enough” test suite [99].

$$\textit{Branch Coverage} = \frac{\textit{Number of branches executed at least once}}{\textit{Number of all branches}} \quad (9.1)$$

A test suite that achieves 100% coverage according to a certain coverage criterion is called an *adequate test suite*. For example, a test suite is branch-adequate when its branch coverage is 100%. However, it is known that statement-adequate or branch-adequate test suites are ineffective for assessing the fault detection capability of a test suite [15, 16, 76, 163]. An alternative test coverage metric that is used mostly in safety-critical context is decision coverage, namely the proportion of decision points triggered by tests. The avionics standard RTCA/DO-178C and the automotive standard ISO26262 enforce complete modified condition/decision coverage (MC/DC) [13, 14]. Writing tests that achieve complete MC/DC is very difficult, and needs a high level of expertise in the system under test [164]. Yet, even 100% MC/DC does not guarantee the absence of faults [165, 166].

9.2.3 Mutation Testing

Mutation testing is the process of injecting faults into a software system and then verifying whether the test suite indeed fails (i.e. detects the injected fault). The idea of mutation testing was first mentioned in a class paper by Lipton (as reported by Offutt et

al. in [30]), and later developed by DeMillo, Lipton and Sayward [38]. The first implementation of a mutation testing tool was done by Timothy Budd in 1980 [39].

Mutation testing induces the following steps on the test process. It starts with a *green* test suite — a test suite in which all the tests pass. First, a faulty version of the software is created by introducing faults into the system (*Mutation*). This is done by applying a known transformation (*Mutation Operator*) on a certain part of the code. After generating the faulty version of the software (*Mutant*), it is passed on to the test suite. If there is an error or failure during the execution of the test suite the mutant is marked as killed (*Killed Mutant*). If all tests pass, it means that the test suite could not catch the fault and the mutant has survived (*Survived Mutant*).

Invalid Mutants.

In the process of generation of mutants, sometimes a mutant is not compilable. Such mutants are called *invalid mutants*. Given the fact that typical mutation testing tools do not attempt to compile the code entirely, it is possible that mutants are created that adhere to the syntax of a language, but cannot be compiled. For example, in case of concatenation of two string variables using “+” operator, changing this operator to “-” leads to generation of an invalid mutant. While most invalid mutants can be avoided at mutant generation time, some are difficult to filter out without having the facilities of a compiler.

Equivalent Mutants.

If the output of a mutant for all possible inputs is the same as the original program, it is called an *equivalent mutant*. It is not possible to create a test case that passes for the original program and fails for an equivalent mutant, because the equivalent mutant has the same semantics as the original program. This makes the creation of equivalent mutants undesirable, since the time that the developer wastes on an equivalent mutant does not result in the improvement of the test suite. Equivalent mutants have a significant impact on the accuracy of the mutation coverage [40]. Unfortunately, equivalent mutants are not easy to detect because they depend on the context of the program itself [41]. For example, in Figure 9.1, -- replacing ++ in *proc1* changes the output for any input other than 0, while the same mutant in *proc2* does not. Indeed, the preceding line $i++$ ensures that the condition $i > 0$ is always met for $i \geq 1$. The mutant can be killed in *proc1*, because $i = 0$, however in *proc2* the mutant is undetectable by any test because of $i = 2$.

As for filtering the equivalent mutants, there are no tools available that automatically detect and remove all equivalent mutants. In general, detection of equivalent mutants is an undecidable problem [42]. Manual inspection of all mutants is the only way of filtering all equivalent mutants, which is impractical due to the amount of work it needs. There-

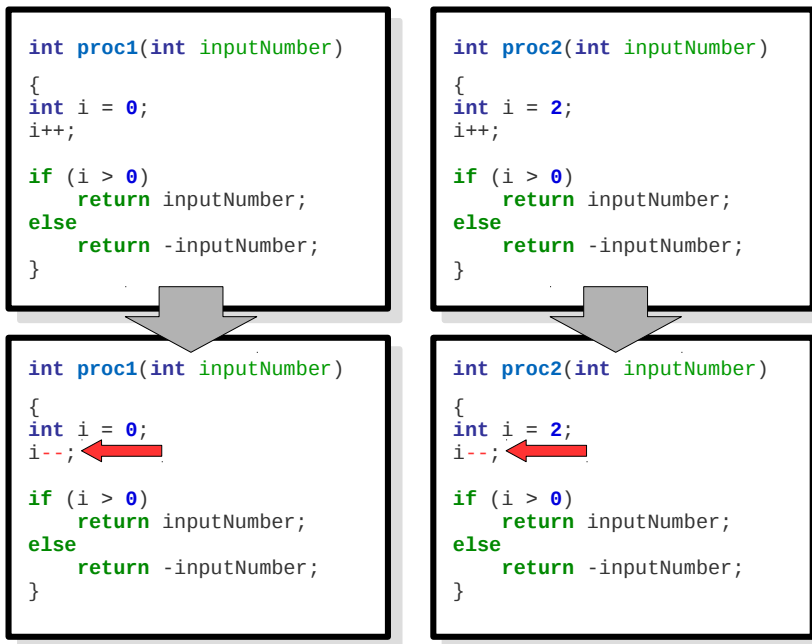


Figure 9.1: Example of an equivalent mutant in proc2

fore, the common practice within today’s state-of-the-art is to take precautions to remove as many equivalent mutants as possible (e.g. using Trivial Compiler Equivalence [43]), and accept equivalent mutants as a threat to validity.

Mutation Coverage.

Mutation testing allows software engineers to monitor the fault detection capability of a test suite by means of *Mutation Coverage* (see Equation 9.2). A test suite is said to achieve *full mutation test adequacy* whenever it can kill all of the non-equivalent mutants, thus reaches a mutation coverage of 100%. Such test suites are called *mutation-adequate test suites*.

$$Mutation\ Coverage = \frac{Number\ of\ killed\ mutants}{Number\ of\ all\ non-equivalent\ mutants} \quad (9.2)$$

Mutation coverage is often declared as a *stopping criterion* for writing (unit) tests — the next level of testing can only start when mutation coverage exceeds a given threshold [44, 45]. This is especially useful when tests are generated automatically [46, 47].

Mutation Operators.

A mutation operator is a known transformation which creates a faulty version by introducing a single change. The first set of the mutation operators designed were reported in King et al. [48]. These operators which work on very basic entities were introduced in the tool Mothra which was designed to mutate FORTRAN77 programming language. In 1996, Offutt et al. determined that a selection of few mutation operators are enough to produce similarly capable test suites with a four-fold reduction of the number of mutants [49]. This reduced set of operators shown in Table 9.1 remained more or less intact in all subsequent research papers.

Table 9.1: Reduced-set mutation operators (adapted from [1] ©ACM 2006)

Operator	Description
AOR	Arithmetic Operator Replacement
AOD	Arithmetic Operator Deletion
AOI	Arithmetic Operator Insertion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COD	Conditional Operator Deletion
COI	Conditional Operator Insertion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOD	Logical Operator Deletion
LOI	Logical Operator Insertion
ASR	Assignment Operator Replacement

With the popularity of the object-oriented programming paradigm, there was a need to design new mutation operators to simulate the faults that occur in this kind of programs. Several studies proposed new mutation operators [50, 51], and some of them were designed to prove the usefulness of object-oriented operators [52, 53]. Ahmed et al. did a complete survey on this subject [54].

During the past decade, the academic focus was on creating new mutation operators for special purposes such as targeting certain security problems [55, 56] or language specific mutation operators [36, 37, 57, 58, 59]. These mutation operators, even though important in their own context, do not relegate into the general concept of mutation testing. The traditional mutation operators are by far the most often implemented [23]. One reason for this is that using more mutation operators produces more mutants; which makes the procedure longer to finish, and as a result, less practical. The reduced set of operators mentioned in Table 9.1 provides a smaller set which produces results with enough detail for any practical purpose, even though the confidence in such results are slightly less than those retrieved by using additional mutation operators.

Mutant Sampling.

To make mutation testing practically applicable, it is important to reduce the time needed — do fewer, do smarter, and do faster [30]. “Do fewer” is achieved by *mutant sampling*: randomly selecting a sample set of mutants instead of processing all of them. This idea was first proposed by Acree [60] and Budd [39] in their PhD theses. Since then, there were many studies confirming the effectiveness of this approach: the performance gain is significant yet reveals the same weaknesses [48, 61, 62, 63]. The random mutant selection can be performed uniformly, meaning that each mutant has the same chance of being selected. Otherwise, the random mutant selection can be enhanced by using heuristics based on the source code.

The percentage of mutants that are selected determines the *sampling rate* for random mutant selection. Using a fixed sampling rate is common in literature [63, 64, 65]. However, it is possible to use a weight factor to optimize the sampling rate according to various parameters such as the number of mutants per class. This is called *weighted* mutant sampling [33]. It is also possible to determine the sampling rate dynamically while performing mutation testing. A method resembling the latter was proposed by Sahinoglu and Spafford to randomly select the mutants until the sample size becomes statistically appropriate [66]. They concluded that their model achieves better results due to its self-adjusting nature [31].

There is one other factor besides the sampling rate that needs to be considered when sampling; the total amount of time that is practically viable. Unfortunately, in the current literature we did not find any concrete targets. Therefore, we set our own target based on a hypothetical scenario of an agile team running the whole mutation testing once every week during the weekend. In this scenario, the team works from Monday at 8am till Friday at 6pm, which leaves the whole weekend (thus 62 hours) to perform the analysis.

9.3 TOOLS USED IN THIS STUDY

In this section we present the test coverage tools used to investigate the trade-offs between branch coverage and mutation coverage in an industrial setting. These tools are JaCoCo, a tool to compute statement and branch coverage for Java software systems (Section 9.3.1); PITest a state-of-the-art mutation testing tool designed for easy integration with current test and build tools (Section 9.3.2) and LittleDarwin, the tool we adapted and used to perform mutation testing on Java software systems with complicated build systems (Section 9.3.3).

9.3.1 JaCoCo

JaCoCo [<http://www.eclemma.org/jacoco>] is a lightweight, flexible, and well documented tool to provide statement and branch coverage for Java programs. JaCoCo is compatible with most Java build systems, hence is easily deployable in a continuous integration environment. JaCoCo is the de facto standard for measuring test coverage for Java projects, and is used as baseline in research concerning test coverage (e.g. [167, 168, 169, 170]). JaCoCo uses a set of different probes to calculate coverage metrics. All these probes are instrumented into Java class files which are Java byte code instructions and debug information optionally embedded therein. Consequently, JaCoCo uses dynamic analysis to compute coverage over byte code which allows it to work even without the source code available. The link to source files are then generated using the debug information that accompanies Java byte code.

However, byte code instrumentation has known disadvantages [167, 171]: By performing a study on JaCoCo, Tengeri et al. identify 6 reasons why using byte code instrumentation in Java language might not be as accurate as source code instrumentation [171]. In particular, (i) the act of instrumentation itself can affect the behavior of the tests, (ii) cross-coverage among submodules is not taken into account, (iii) untested submodules are excluded from the analysis, (iv) method signatures are different in byte code and source code for methods that contain compiler-injected parameters, (v) exceptions leading to interruption of the control flow result in loss of information, and (vi) generated code produces obstacles in collecting coverage information. Consequently, the results from JaCoCo—especially when statements or branches are reported as *not* being covered by a test—need to be double-checked for accuracy.

JaCoCo is used throughout this study to compute branch coverage for industrial and open source cases.

9.3.2 PITest

PITest [<http://pitest.org/>] is a state-of-the-art mutation testing system for Java, designed to be fast and scalable. PITest seamlessly integrates with today's test and build tools (i.e. Ant, Gradle and Maven). PITest is the de facto standard for mutation testing within Java, and it is used as baseline in research concerning mutation testing (e.g. [76, 77, 172, 173, 174]).

PITest has a wide range of mutation operators. The default setting is practically a subset of the reduced-set of mutation operators (Table 9.2), however because they are applied to byte code, they are grouped and named differently. There are several other mutation operators that can be enabled as well. PITest comes with a lot of internal optimizations

to tackle the “do faster” part of the maxim: *do fewer, do smarter, and do faster* [30]. Most importantly, mutations are performed at the level of byte code to avoid recompilation. However, byte code level mutation presents other obstacles to overcome. For example, PITest needs to find and execute tests by itself for each mutant, which causes incompatibility with complicated build structures. This is more apparent when test code is located in separate packages, and therefore, not easily discoverable by PITest. In addition, PITest incorporates some heuristics to choose which tests to run, which implies that the accuracy of these heuristics affects the results of PITest as well. In particular, PITest uses the same mechanism as JaCoCo to determine statement coverage, and skip the evaluation of mutants in uncovered statements. This, in turn, raises similar issues as described in Section 9.3.1.

PITest is used in this study in an attempt to demonstrate the feasibility of mutation testing in an industrial environment (RQ1).

Table 9.2: PITest mutation operators at the time of this study

Operator	Description	Example	
		Before	After
CBM	Mutates the boundry conditions	$a > b$	$a \geq b$
IM	Mutates increment operators	$a ++$	$a --$
INM	Inverts negation operator	$-a$	a
MM	Mutates arithmetic & logical operators	$a \& b$	$a b$
NCM	Negates a conditional operator	$a == b$	$a != b$
RVM	Mutates the return value of a function	return true;	return false;
VMCM	Removes a void method call	voidCall(x);	—

9.3.3 LittleDarwin

LittleDarwin [<http://littledarwin.parsai.net/>] is a mutation testing tool created by Ali Parsai (first author of this paper) to provide mutation testing within a continuous integration environment. It is designed to have a loose coupling with the test infrastructure, instead relying on the build system to run the test suite. LittleDarwin imposes two restrictions only: (a) the build system must be able to run the test suite; (b) the build system must return non-zero if any tests fail, and zero if it succeeds. For a detailed description of LittleDarwin, please refer to Parsai et al. [32].

For the purposes of this study, there are 9 mutation operators implemented in LittleDarwin listed Table 9.3. These operators are a subset of the reduced-set of mutation operators (Table 9.1), and similar to the default setting of PITest, but differently grouped and named. Since the number of mutation operators of LittleDarwin is limited, it is possible that no mutants are generated for a class. In practice, we observed that usually

only very small compilation units (e.g. interfaces, and abstract classes) are subject to this condition.

Table 9.3: LittleDarwin mutation operators

Operator	Description	Example	
		Before	After
AOR-B	Replaces a binary arithmetic operator	$a + b$	$a - b$
AOR-S	Replaces a shortcut arithmetic operator	$++a$	$--a$
AOR-U	Replaces a unary arithmetic operator	$-a$	$+a$
LOR	Replaces a logical operator	$a \& b$	$a b$
SOR	Replaces a shift operator	$a >> b$	$a << b$
ROR	Replaces a relational operator	$a >= b$	$a < b$
COR	Replaces a binary conditional operator	$a \&\& b$	$a b$
COD	Removes a unary conditional operator	$!a$	a
SAOR	Replaces a shortcut assignment operator	$a * = b$	$a / = b$

For the moment, LittleDarwin is not optimized for speed. For each mutant injected, LittleDarwin demands a complete rebuild and test cycle on the build system. This easily leads to several hours of analysis time. Currently the only way to speed-up LittleDarwin is to use mutant sampling, which is covered under RQ3.

LittleDarwin is used throughout this study to compute mutation coverage for industrial and open source cases.

9.4 CASE STUDY DESIGN

In this section, we explain the details about the setup of our case study. We start describing the industrial case (Section 9.4.1) and open source cases (Section 9.4.2). Then, we report the setup of the tools (in Section 9.4.3). Finally, we provide an overview of the comparison criteria (in Section 9.4.4).

9.4.1 Industrial Case

There are three main components which create the core of Impax ES Clinical Applications. One of these three components is the Segmentation component. The main use of this component is to provide imaging algorithms to segment 3D volumes. This component is average in size compared to the other components of the system, and it includes a test suite which was under active development at the time of the case study. The component is entirely written in Java. The team is geographically dispersed across four cities around the globe, thus making coordination a critical part of the software development. This also increases the importance of the test suite during build, because faults discovered downstream require long distance communication over different timezones. The

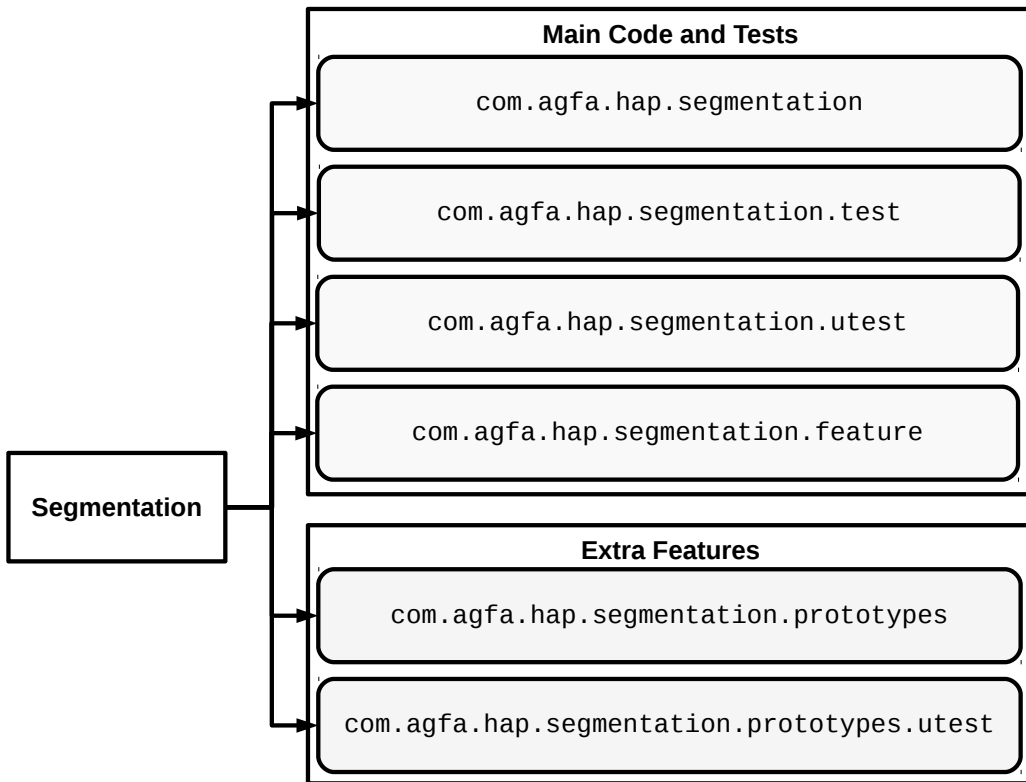


Figure 9.2: Agfa HealthCare Segmentation build components

team uses the SCRUM model of development, by holding a weekly sprint meeting to coordinate their efforts [175]. They use a typical continuous integration with separate build servers and source code repository servers, centered around Maven. The build configuration includes several plugins to compute code coverage, generate reports, and obfuscate the target classes.

The Impax ES system is released in two main variants (production or prototype). There are a few minor variants as well, depending on the target hardware platform the system is deployed upon. Therefore, the Maven build system was configured to resolve dependencies with libraries depending on the product line variant to be built. The system architecture itself relies on the OSGI (Open Service Gateway Initiative) dynamic component model, to load and unload components dynamically without rebooting the system. The extensive use of (dynamic) OSGI headers implies a complicated build process where the Maven plug-in Tycho [<https://eclipse.org/tycho/>] is used to fetch dependencies, compile source files, and run the test suite.

The Segmentation component itself is divided into 6 subcomponents as shown in Fig-

ure 9.2. The main code is located in *com.agfa.hap.segmentation* while the test suite code, some resources used by unit tests and a feature manifest for eclipse are located in the *com.agfa.hap.segmentation.{utest, test, feature}* subcomponents respectively. The other two subcomponents provide early prototype features and their unit tests which should be excluded from the final build. Each subcomponent has a separate *pom.xml* file and can be built on its own. There is also a parent *pom.xml* file which selects and builds these subcomponents based on the profile of the product-line variant to be built (production or prototype).

For our case study, we focus on unit test suite of Segmentation component, and we do not include the acceptance tests. Removing the acceptance tests decreases the total time for compilation and testing of the Segmentation component from a few hours to less than a minute in each build.

9.4.2 Cases Under Investigation

To increase the generalizability of our findings we analyzed four open source systems for addressing RQ2. The descriptive statistics of all cases under investigation (the industrial one + the four open source ones) are reported in Table 9.4. Because the non-disclosure agreement does not allow us to reveal too many details, we only list approximate statistics for the industrial case.

Table 9.4: Descriptive statistics for cases under investigation

Case	URL	Version	Size (LoC)		Ratio
			Main	Test	
<i>Industrial Case</i> (The numbers are approximated for confidentiality reasons.)					
Agfa Segmentation	http://www.agfahealthcare.com/global/en/main/resources/product_images/impax_6_0.jsp	3.7-snapshot	38K	50K	~1.3
<i>Open Source Cases</i>					
Joda Time	http://www.joda.org/joda-time/	2.8	28479	54645	1.92
Apache Commons Codec	http://commons.apache.org/proper/commons-codec/	1.7	5773	9917	1.72
jOpt Simple	http://pholser.github.io/jopt-simple/	5.0	1958	6072	3.10
AddThis Codec	http://github.com/addthis/codec	3.2.1	3614	1318	0.36

9.4.3 Tool Setup

JaCoCo was already in use as part of the Maven build configuration in the industrial case and some of the open source cases, therefore we used JaCoCo to calculate the branch coverage for the rest as well. For this case, JaCoCo was being used with its default parameters. PITest was run using the default suite of mutation operators, and it was run in parallel mode, which detects the number of available CPU cores and uses all of them for the analysis. Finally, LittleDarwin was run with two sets of commands: the first to compile the mutated source code and install the final result into the local Maven repository; the second to execute the test suite on the compiled version. This was necessary because the production code and the test code have separate build systems.

To make the performance comparison (RQ3) valid, we ran the analysis of all open source cases on the same machine, operating system, and python interpreter. However, since the industrial case was analyzed 5 months prior to the open source cases on premise, the analysis was done on different hardware and operating system, hence the absolute numbers of the execution times cannot be compared. For the industrial case, the machine used has two Intel Xeon 2.80 GHz processors with 16 GB (4x4 GB) of DDR3-1333 memory running Windows 7 Enterprise Edition. Since LittleDarwin only uses a single thread to perform its analysis, there are no gains from the multi-core architecture of the hardware it runs on. The open source cases were analyzed on a custom made PC with AMD 1090T 3.2 GHz processor and 8 GB (2x4 GB) memory running Linux Mint 17. The execution time for a build is extracted from the output of the build system (which was Maven in all our cases). The execution time for JaCoCo was extracted in the same manner, since JaCoCo acts as a Maven plugin. The execution time for LittleDarwin was calculated by aggregating the time spent generating the mutants and the time spent for gathering the results of the execution of the tests for all mutants.

9.4.4 Comparison Criteria

In this paper we analyze branch and mutation coverage from a conceptual point of view. For both metrics, a higher value is assumed to suggest a good test suite quality to the developer. While branch coverage is widely used in industry, mutation coverage is known to be a better indication of fault detection capability of a test suite. For this reason, we are interested to explore the situation where branch and mutation coverage present different values.

Considering m as mutation coverage percentage, b as the branch coverage percentage, and t as a threshold, we define 5 categories (Table 9.5). Note that we consider coverage at class level, hence the unit of analysis is a *class*.

Table 9.5: Categorization of differences between branch and mutation coverage

Category	Definition	Description
SimCov	$(m - b \leq t \wedge m, b > 0) \vee m = b = 0$	Similar branch and mutation Coverage
LoB-HiM	$m - b > t \wedge m, b > 0$	Low Branch coverage and High Mutation coverage (confirms the fault detection capability of the test suite)
HiB-LoM	$b - m > t \wedge m, b > 0$	High Branch coverage and Low Mutation coverage (false confidence regarding the fault detection capability of the test suite)
NoB	$m > 0 \wedge b = 0$	No Branch coverage
NoM	$b > 0 \wedge m = 0$	No Mutation coverage

- The Category SimCov corresponds to the category in which the difference between m and b is less than a given threshold ($t\%$). For these classes mutation coverage does not provide extra information with respect to branch coverage.
- For the category LoB-HiM the mutation coverage is larger than the branch coverage. There mutation testing provides extra confidence concerning the test suite; despite the low branch coverage the test suite has a high fault detection capability. *This category represents those classes where branch coverage is “good enough”.*
- In contrast, the category HiB-LoM marks classes where the mutation coverage is smaller than the branch coverage. This is the most interesting category for our investigation. Indeed, mutation testing reveals weaknesses in the test suite, namely where test suite lacks of detectability of a potential fault (Section 9.2.1). From another point view, *this category shows where high branch coverage gives a sense of false confidence regarding the fault detection capability of the test suite.*
- Finally, the categories NoB and NoM mark the special cases where the corresponding coverage metric is zero. If both the branch and mutation coverage are zero, it most likely corresponds to a class which is never tested. However, for NoB, this may also be due to anomalies in the byte code level instrumentation of JaCoCo (see Section 9.3.1). When NoM is zero this is most likely caused by lack of mutable statements in the code (see Section 9.3.3).

The value of t determines the threshold that the two coverage scores are considered close enough so that the difference between them does not make any practical difference. For example, for a threshold of 10%, if for a particular class the branch coverage is 55% and mutation coverage is 63%, having either of these scores does not change the perceived test quality of that class, and the coverage can be interpreted as “around 60%”. Therefore, by adjusting t , we can model the sensitivity of developers to the coverage score. This threshold does not have any effect on the number of classes in Categories NoB and NoM, and only changes the number of classes in Categories SimCov, LoB-HiM and HiB-LoM. To determine t we examine the results of both metrics, and choose the minimum value of t where neighboring t values would not change the number of classes in each category. This way, the threshold is selected based on the specifics of each case, minimizing the

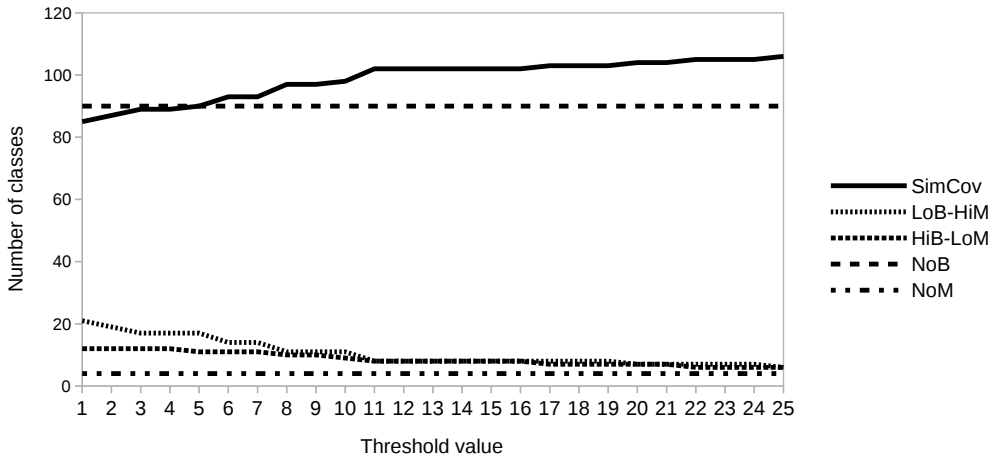


Figure 9.3: Number of classes in each category for t values between 1 and 25 in the industrial case

analysis bias. For our investigation, we tried all ordinal values of t between 1 and 25 and ultimately derived $t = 11$ and $t = 7$ respectively for the industrial and the open source cases. The result of this derivation for the industrial case is shown in Figure 9.3. The number of classes in different categories remain fairly constant after $t = 11$, hence is chosen as the threshold.

Another way to compare mutation and branch coverage is to compute the correlation of these two metrics. Specifically, we analyze whether the order of a set of classes by one metric can predict the order by another set using Kendall’s τ_b coefficient as described in statistic handbooks [103, 104]. This has been used previously in a similar study by Gligoric et al. where they argue that this coefficient is more appropriate than a linear correlation coefficient because it does not assume a linear correlation between the metrics [74].

9.5 RESULTS AND DISCUSSION

In this section, we discuss the results of our study. For each research question, we briefly describe our motivation, then our approach, and conclude with our observations.

9.5.0 RQ1. *Is It Feasible to Integrate Mutation Testing in A Continuous Integration System?*

Motivation. Mutation testing adds the following steps to the build process: (a) inject a mutant into the software system, (b) build the mutated system, (c) execute the test suite, (d) record the results, (e) restore the system to its original state, and (f) repeat the procedure until there is no more mutants. These steps easily interfere with other parts of the build process (i.e. selection of product line variants, dependency resolution, and dynamic component loading), hence the reason to address the feasibility issue.

Approach. To answer this question we follow a proof by construction. We integrate a state-of-the-art mutation coverage tool (namely PITest) into the build environment of the industrial case (namely Maven). The industrial case is representative for many other industrial systems: it has legacy components (where unit tests are missing), it has two major variants (incorporated in a product-line architecture), and has a complicated build structure (where components are dynamically loaded into the build environment by means of OSGI). We report the challenges we encountered and the workarounds we performed, all to no avail. Consequently, we adapted and used a mutation testing tool named LittleDarwin specifically designed to integrate well within a continuous integration environment.

Findings. We considered a series of tools during our feasibility study: PITest, Cheshire, and MuUnit. PITest is a widely-used mutation testing tool aimed at industrial projects; Cheshire is a tool to convert OSGI interfaces and MuUnit is a mutation testing tool designed to perform its analysis on OSGI projects. The challenges we faced performing the proof by construction are summarized as follows:

- **OSGI.** As explained in Section 9.4.1, the industrial case heavily relies on the OSGI (Open Service Gateway Initiative) dynamic component model for dynamic loading and unloading of components. We first considered to refactor the system and remove the OSGI headers. However, we learned that these OSGI headers are deeply embedded in all of the source code. Removing the OSGI headers would alter the code beyond recognition hence was not an option.
- **PITest.** PITest does not refer to OSGI in its documentation, nor did we find any other information sources. We tried it out ourselves, and quickly discovered that PITest could not run the OSGI-dependent code by itself, and the Tycho plug-in for Maven was incompatible as well. We posted a few questions in the PITest forums and there it was confirmed that OSGI could cause problems (see <https://groups.google.com/d/topic/pitusers/IH21Q4jJaco/discussion>). In particular, there were two blocking issues that we encountered during our attempt. First,

PITest cannot handle a test suite in a completely separated package, loaded dynamically via OSGI and/or Tycho. Second (and related to the first issue), the PITest test selection heuristics deciding which tests should be run first (see Section 9.3.2) could not find the tests due to dynamic loading of components.

- **Cheshire.** [<http://github.com/AlFranzis/cheshire>] As the next option, we explored the possibility to automatically convert the project into a non-OSGI one during the build. Cheshire is a prototype tool that provides an interface for OSGI-compliant software to resolve and retrieve dependencies during compile time. After e-mail communication with the developer of Cheshire, it was clear that many extra recipes should be written for the various kinds of dependencies used within the industrial case. According to the developer's estimation, it would take weeks for someone not familiar with the details of the component. Even then, the lack of previous experience with a combination of PITest and Cheshire made the final result unpredictable. Therefore, this solution was dismissed.
- **MuUnit.** [<https://code.google.com/p/muunit>] Finally, we tried to incorporate an OSGI-compliant tool to perform the mutation testing. The only prototype suitable for this task was MuUnit [176]. After a quick try-out it became clear that at the time of our analysis (which was September 2014) MuUnit was an early prototype able to run its analysis on simple projects only. Since then, there has been no development on this project, and as it stands, it can be considered an abandoned project. For this reason, this solution was dismissed.
- **Others.** We considered two other options, Jumble [<http://jumble.sourceforge.net/>] and Javalanche [<http://www.st.cs.uni-saarland.de/mutation/>]. A cursory analysis of these tools and their documentation revealed that the OSGI components would cause similar problems as we had with PITest, hence these options were dismissed as well.

Lessons Learned. This feasibility study demonstrated that—contrary to common wisdom—it is not that easy to integrate mutation testing into a complicated build process. This is caused by the interference between the mutation testing (deeply coupled with the test infrastructure in order to speed up the process) and the product line configuration (with dynamic loading of test components). As often within software engineering, it is an accidental problem not an essential one [177]. Indeed, if the development team of the tools under investigation would choose to do so, they could probably engineer a solution. Yet, at the time of analysis the OSGI headers were too deeply embedded in the case under investigation to be handled by the tools available.

Due to aforementioned problems, Ali Parsai (the first author of the paper) developed and adapted a proof-of-concept tool called LittleDarwin, described in Section 9.3. Lit-

tleDarwin is explicitly designed to be loosely coupled to the test infrastructure, completely relying on the build system to run the tests. However, in doing so LittleDarwin forsakes the speed-up enabled by deep analysis of the test infrastructure and fast mutation injection via byte-code manipulation. We successfully applied LittleDarwin to perform the mutation testing on Segmentation. The problems we encountered using other tools were alleviated by the fact that LittleDarwin itself does not run the tests, but rather, the build system does. Therefore, the build system fetches the OSGI dependencies, and creates and configures the test harness. By using LittleDarwin to analyze Segmentation successfully, we demonstrated the feasibility of mutation testing within an industrial continuous integration environment.

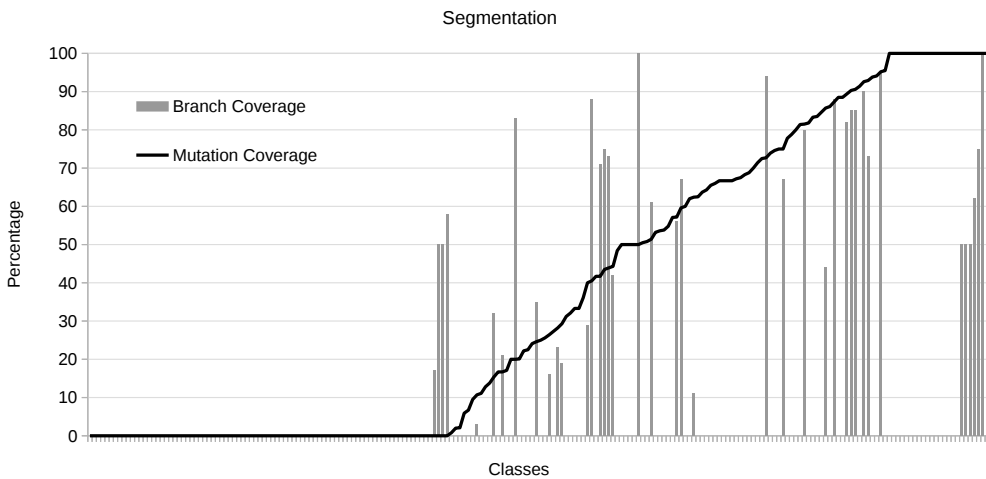
RQ1 Summary

Due to the accidental complexity, byte-level mutation tools such as PITest cannot easily be integrated into a complicated build process. Yet, if one decouples the mutation tool from the test infrastructure (thus relies on the build system to execute the tests) it is feasible to integrate mutation testing in a continuous integration setting. However, one does so at the expense of performance, i.e. mutation testing cannot be done as smartly and efficiently as the tightly coupled counterpart.

9.5.0 RQ2. Does Mutation Testing Reveal Additional Weaknesses in the Test Suite Compared to Branch Coverage?

Motivation. On the one hand, mutation testing has been demonstrated to subsume branch coverage [26, 152]. On the other hand, Gligoric et al. reports that among several coverage criteria, branch coverage is the best one to predict the mutation coverage of a test suite [74]. In this context, analyzing them together helps to determine whether or not mutation testing is capable of highlighting where branch coverage offers false confidence on the quality of the test suite.

Approach. Just as with RQ1, the unit of analysis is a class. We collect branch coverage via JaCoCo and mutation coverage via LittleDarwin for one industrial system and four open source systems listed in Table 9.4. For each class in the systems under study, we analyze the branch and mutation coverage, classifying them in the five categories shown in Table 9.5. By focusing on code sections characterized by the difference between branch coverage and mutation coverage, we show how mutation coverage exposes additional weaknesses. The similarity threshold we derived was $t = 11$ for the industrial case and $t = 7$ for the open source cases. We also calculate the Kendall correlation coefficient (τ_b) and the p-value.



[The horizontal axis represents the classes sorted by mutation coverage; the vertical axis is the coverage percentage. Each bar represents the branch coverage for a class, and each point on the line represents the mutation coverage for a class.]

Figure 9.4: Visual comparison of branch coverage (JaCoCo) versus mutation coverage (LittleDarwin) on the industrial case

Findings for the Industrial Case. The industrial case has 212 classes. During the analysis, 4,955 of the 12,825 generated mutants were killed by the test suite, resulting in 38.6% overall mutation coverage. Figure 9.4 shows the mutation coverage and branch coverage for each class. First of all, a large number of classes (48%) have zero branch coverage and mutation coverage, illustrating the inadequacy of the test suite. This is true considering that we focus on the unit tests only; indeed, there was a suite of acceptance tests which did exercise most of the code but takes hours to execute. Secondly, there are several classes where the branch coverage and mutation coverage vary by a large margin.

The same data, organized according to the classification described in Section 9.4.4, is reported in Table 9.6. The table lists the number of classes for each of the five categories, as well as Kendall correlation coefficient (τ_b) and the p-value. First of all, we see that for 102 out of 212 classes (thus less than half) the branch and mutation coverage are the same for all practical purposes. For these classes, mutation testing does not provide additional value. Secondly, there are 8 classes in the category LoB-HiM, where the mutation coverage is larger than the branch coverage. Given the fact that LittleDarwin includes ROR mutation operator, at first sight it is expected that wherever there is mutation coverage, it should guarantee branch coverage. However, with deeper analysis we found this not to be true because there are lots of multi-branched methods that include many arithmetic operations only in a few of branches. The tests often target only these branches due to their

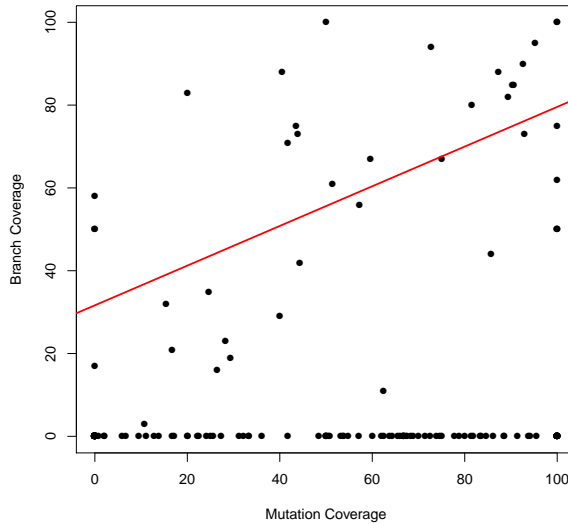
Table 9.6: Comparing branch coverage (JaCoCo) versus mutation coverage (LittleDarwin) on cases under study

Case	Categorization					Correlation	
	SimCov	LoB-HiM	HiB-LoM	NoB	NoM	Kendall τ_b	p-value
<i>Industrial Case</i>							
Segmentation	102	8	8	90	4	0.25	8.951×10^{-6}
<i>Open Source Cases</i>							
Joda Time	43	18	12	0	70	0.11	1.132×10^{-1}
Apache Commons Codec	27	3	7	0	0	0.31	1.755×10^{-2}
jOpt Simple	30	2	1	0	1	0.71	2.863×10^{-6}
AddThis Codec	9	9	8	0	0	0.53	2.146×10^{-4}
Total	211	40	36	90	75	N/A	

perceived importance, and due to the large number of arithmetic mutants generated for these branches, there is an abnormal inflation of mutation coverage. Here, mutation testing provides extra confidence concerning the test suite; despite the low branch coverage, the test suite has a high coverage over fault-prone areas of the code. For these classes, mutation testing provides additional value. Most interestingly, there are 8 classes in the category HiB-LoM, where the mutation coverage is smaller than the branch coverage. There the mutation testing reveals weaknesses in the test suite; the high branch coverage gives a sense of false confidence regarding the test suite, even though the mutation coverage shows that some covered branches are indeed not adequately tested.

The most extreme case in this respect is the class *Discrete3DContour*, which has 88% branch coverage yet only 41% mutation coverage. For this class additional tests are needed, since the current tests cannot reveal injected faults. Most surprisingly (also apparent in Figure 9.4), there are 90 classes with zero branch coverage, yet some mutation coverage (Category NoB). One representative example is the class *RegionGrowerNeighbours*; which has 0% branch coverage yet all 22 generated mutants are killed by the *VolumeGrowerTest* and *RegionGrowerNeighboursTest*. Both tests indirectly verify the algorithms provided by *RegionGrowerNeighbours*, and for this reason it is unlikely that the branch coverage is 0%. Manual inspection confirmed that here as well it was the dynamic loading of components by means of OSGI which leads to the loss of execution traces and thus results in the miscalculation of the branch coverage in JaCoCo (see Section 9.3.1). Finally, there were four classes with zero mutation coverage yet significant branch coverage (58%, 50%, 50% and 17% respectively). There the branch coverage creates an even higher sense of false confidence: there is branch coverage, yet the tests fail to reveal any faults.

Figure 9.5 shows the weak correlation between branch and mutation coverage values of all classes in the industrial case. While correlation cannot be ruled out (Kendall correlation of 0.25, p-value < 0.01), it shows that branch and mutation coverage correlate weakly at best. This may be partially attributed to the 90 classes with zero branch cover-

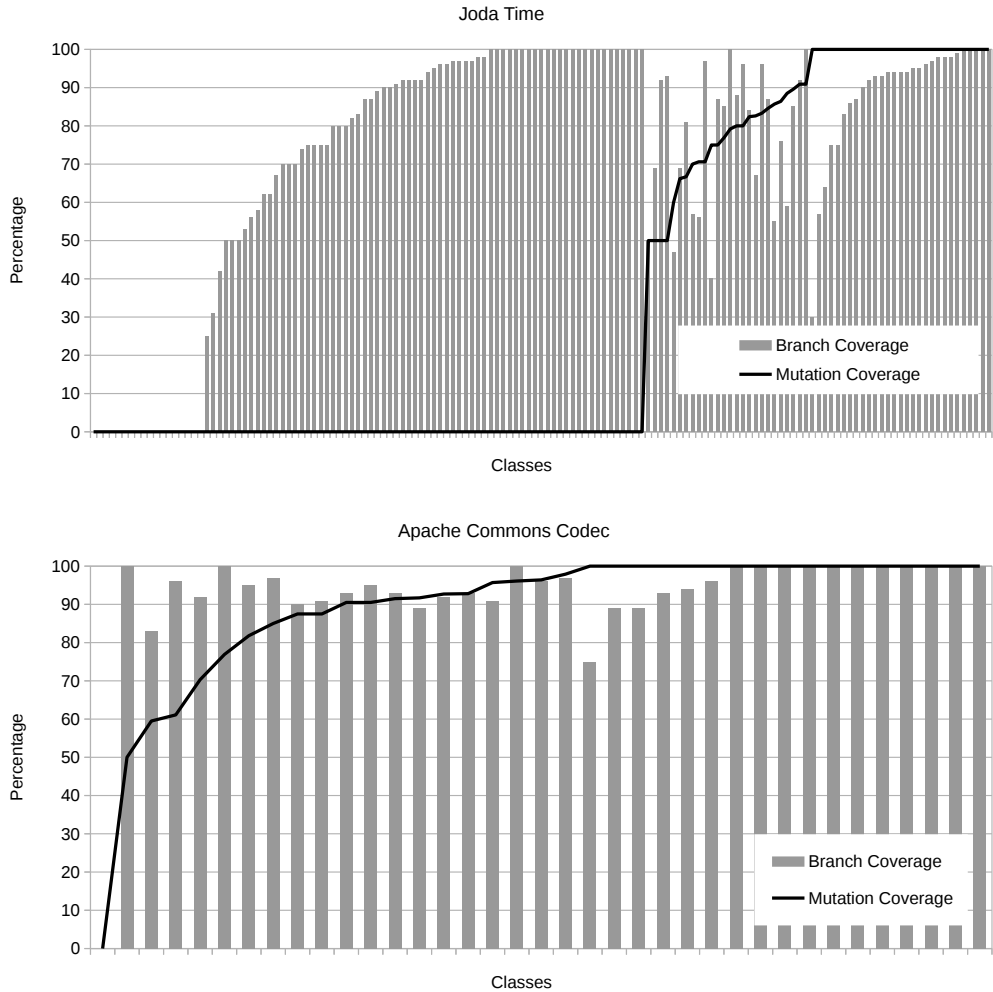


[The horizontal axis represents mutation coverage; the vertical axis represents branch coverage. The red line is the linear regression line.]

Figure 9.5: Weak correlation between branch coverage (JaCoCo) and mutation coverage (LittleDarwin) on the industrial case

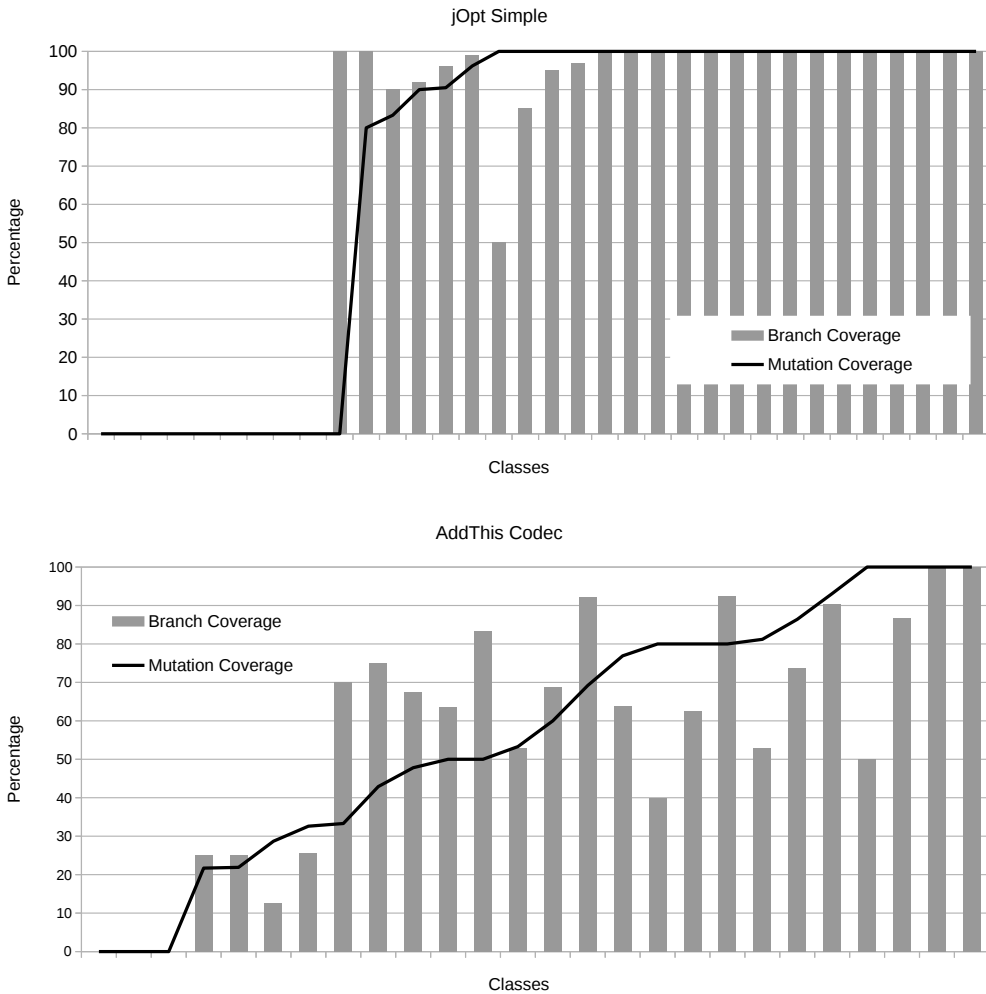
age (the horizontal line of dots at the bottom of the chart), which is an anomaly caused by the dynamic loading of OSGI components.

Findings for the Open Source Cases. Figure 9.6 shows the mutation and branch coverage values for all classes in the open source cases. Most strikingly, the results of the analysis for open source cases are quite different than from the industrial case. This is quite apparent in case of Joda Time, where branch coverage is providing more information than mutation coverage. Moreover, the low number of classes with no coverage at all shows that the open source cases are more adequately tested compared to the industrial case. The same data, organized according to the classification described in Section 9.4.4, is listed in Table 9.6. All four cases show some degree of branch coverage (NoB = 0); three of the four open source cases (Apache Commons Codec, jOpt Simple and AddThis Codec) also have some degree of mutation coverage (NoM \leq 1). Here as well, Joda Time is the outlier (NoM = 70). Manual inspection revealed that a lot of the classes in Joda Time were implemented without any statements that could be mutated by the mutation operators of LittleDarwin (see Section 9.3.3). For example, some data classes include only variables and getter/setter methods, and therefore no mutant is generated for them. This illustrates that it might be worthwhile to expand the mutation operators beyond the



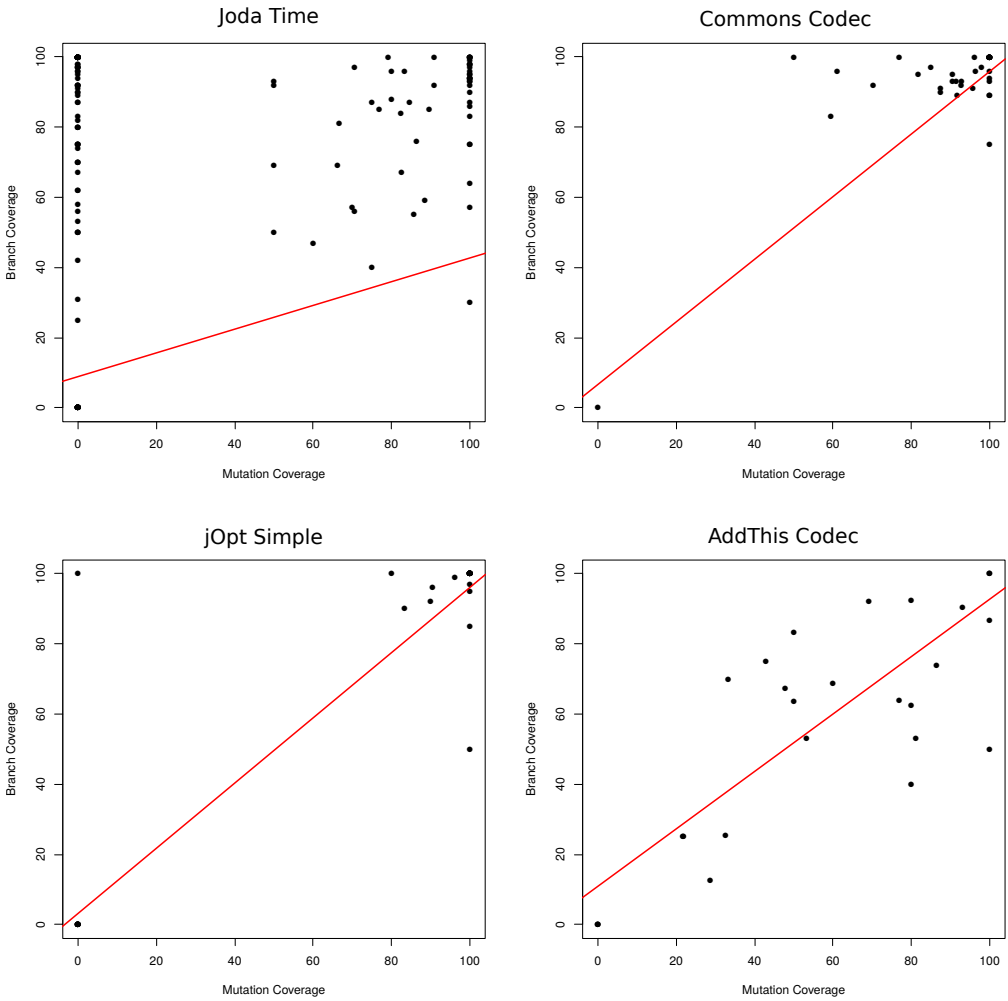
[The horizontal axis represents the classes sorted by mutation coverage; the vertical axis is the coverage percentage. Each bar represents the branch coverage for a class, and each point on the line represents the mutation coverage for a class.]

Figure 9.6: Branch coverage (JaCoCo) and mutation coverage (LittleDarwin) at class level for the open source cases



[The horizontal axis represents the classes sorted by mutation coverage; the vertical axis is the coverage percentage. Each bar represents the branch coverage for a class, and each point on the line represents the mutation coverage for a class.]

Figure 9.6: (Continued from previous page) Branch coverage (JaCoCo) and mutation coverage (LittleDarwin) at class level for the open source cases



[The horizontal axis represents mutation coverage; the vertical axis represents branch coverage. The red line is the linear regression line.]

Figure 9.7: Correlation between branch coverage (JaCoCo) and mutation coverage (LittleDarwin) on the open source cases

reduced set listed in Table 9.1.

Looking at the column SimCov, we see that for most of the classes (ranging from 30% to 88%) the branch and mutation coverage are the same for all practical purposes. Thus for more than half of the classes, mutation testing does not provide additional value. This is quite different from the industrial case, where less than half of the classes had similar coverage values. Nevertheless, there are a significant number of classes in the category LoB-HiM, where the mutation coverage is larger than the branch coverage. Furthermore, there are also a significant number of classes in the category HiB-LoM, where the mutation coverage is smaller than the branch coverage. For three of the four cases, the values in column HiB-LoM are larger than the values in column LoB-HiM and here as well Joda Time is the exception. Thus, although much less than in the industrial case, there are still a significant number of classes where mutation tests reveals additional weaknesses.

Analyzing the correlation between branch and mutation (i.e. columns “Kendall τ_b ” and “p-value”) we see that the correlation between branch and mutation coverage is rather poor. Figure 9.7 provides insight into the lack of correlation. For each of the cases the dots in the scatter plot are in distinct regions, hence the coverage values depend a lot on the particular context of the case under investigation.

Lessons Learned. As seen in Table 9.6, we found branch coverage and mutation coverage of classes to agree only in 47% of them. In 9% of classes, we observed that the density of the mutants were much higher in few branches in the code, and thus despite a low branch coverage, the mutation coverage is much higher. Conversely, in 8% of cases where branch coverage is high and mutation coverage is low, the quality of the test oracles are in question. In the remaining 36% of classes, the peculiarities of the code cause problems in calculation of either metric: In case of branch coverage, complicated structure of Segmentation means that it cannot be accurately calculated, and this is in line with the observations of Tengeri et al. [171]. In case of mutation coverage, the abundance of small data classes or interfaces and stubs in Joda Time means that a more extensive set of mutation operators is required. In total, such analysis on the weaknesses in the tests cannot be done without mutation testing, and it is clear that using branch coverage alone can mislead a developer about quality of the tests.

RQ2 Summary

For all the cases under investigation, we discovered that significant number of classes where mutation testing reveals additional weaknesses compared to branch coverage. However, the added value of the mutation testing is context-dependent and varies between the cases we investigated. This warrants further research into the nature of lower coverage values both for branch coverage and mutation coverage.

9.5.0 RQ3. Can We Reduce the Performance Overhead Induced By Mutation Testing to An Acceptable Level?

Motivation. The results of RQ2 shows that mutation testing provides valuable complementary information over branch coverage. However, in order to avoid issues connected with specificities of the system architecture, the results of RQ1 suggest that we need to do so with mutation testing tools loosely coupled to the test infrastructure, thus inherently slower. For this reason, in this RQ we want tackle in the “time concern” demonstrating that (i) the performance overhead induced by mutation testing can be reduced to meet industrial time constraints and (ii) mutation testing preserves the information presented in RQ2. As a realistic scenario we consider a team that starts working from Monday at 8am till Friday at 6pm. Here, we want to verify whether a complete mutation testing could run once a week during the week-end and right before the sprint meeting scheduled on Monday morning. In this context, we define an acceptable level of performance overhead as a mutation testing job that runs in up to 62 hours, namely between Friday 6pm to Monday 8am. If the full mutation testing cycle takes longer, we use *mutant sampling* to reduce the number of mutants injected into the system, because it has been demonstrated that even with a sample size as low as 50% mutation tests still provides reliable results [63].

Approach. We first measure the performance overhead induced by a full mutation testing of the industrial case (injecting 12,825 mutants for 38K lines of code). Here we calculate the average time for a single iteration by dividing the total time for the process by the number of mutants. Next, we estimate the sample size based on the average single iteration time so that the total analysis time would drop below the 62 hour maximum value. We then compare the coverage values for the full mutation testing and the sampled mutation testing with the same categories as RQ2.

To settle the sampling rate, we used the following procedure. We first analyze the relationship between the number of mutants, and the time required for the analysis. In principle, this should be a linear relationship, which is confirmed in Figure 9.8. Via a linear regression ($\rho = 0.9992$, p-value = 0.000026) we achieve the slope of the regression

CHAPTER 9. COMPARING MUTATION COVERAGE AGAINST BRANCH COVERAGE IN AN INDUSTRIAL SETTING

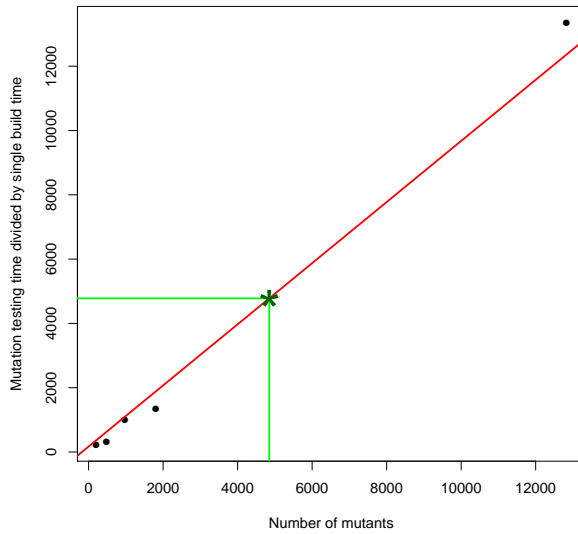


Figure 9.8: Linear relationship between number of mutants and total analysis time

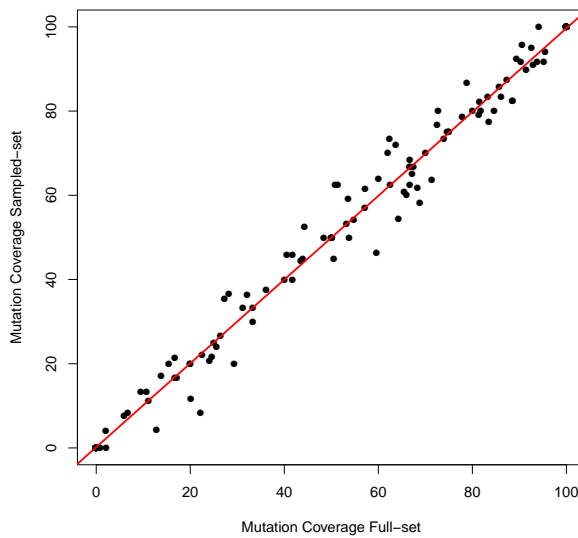
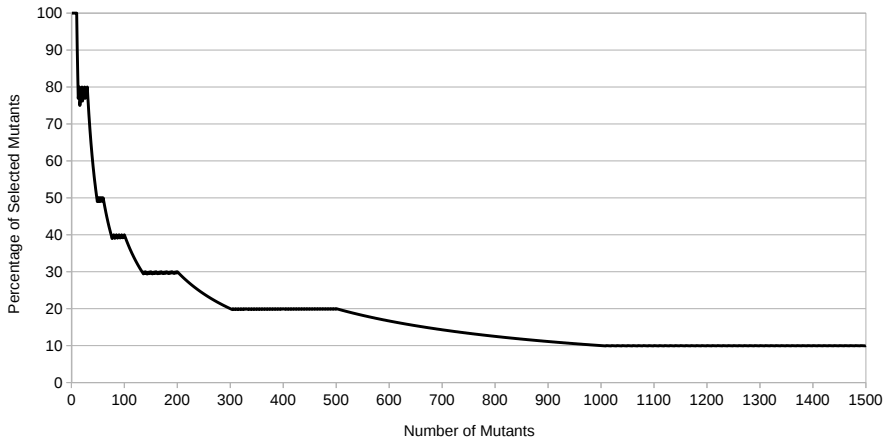


Figure 9.9: Correlation of full mutation coverage against mutation coverage with weighted sampling



[The horizontal axis shows the number of mutants generated for a class;
the vertical axis shows the percentage of selected mutants.]

Figure 9.10: The distribution of the *weighted mutation sampling*

Table 9.7: Comparing branch coverage (JaCoCo) versus mutation coverage (LittleDarwin)—full coverage + weighted sampling)

Case	Categorization					Correlation	
	SimCov	LoB-HiM	HiB-LoM	NoB	NoM	Kendall τ_b	p-value
Industrial Case (full)	102	8	8	90	4	0.25	8.951×10^{-6}
Industrial Case (sampled)	102	10	9	87	4	0.26	4.106×10^{-6}

line as 0.9506 with an offset value of 169.5 (Equation 9.3). Using this equation, we can estimate the upper limit for the number of mutants. The total time needed to perform the analysis is = 62 hours; a single iteration takes 46 seconds, thus we get as an upper limit for the number of mutants 4,780.

$$\text{Number of mutants} = 169.5 + 0.9506 \times \frac{\text{Total analysis time}}{\text{Time for a single iteration}} \quad (9.3)$$

To guarantee that the classes with smaller set of mutants are still represented in the sample set, we used a procedure called *weighted mutation sampling* [33], and we performed the random sampling at class-level rather than project level. The weights are chosen based on the size of the mutant set for each class, thus reduces the masking effect of classes with larger set of mutants in the final coverage score. This way, our sampling method randomly selected 4,448 mutants out of 12,825 generated mutants (34.7%). The percentage of mutants selected based on the size of the mutant set is shown in Figure 9.10.

Findings. The full mutation testing of the industrial case (i.e. the Segmentation component) takes 163 hours to complete; way more than the established upper limit of 62 hours. Via the weighted sampling method, we reduced the number of mutants from 12, 825 to 4, 448 (34, 6%). With this reduction, the time required to perform the mutation testing dropped to 58 hours (almost one-third of the full analysis) well under the established upper limit.

Out of the 4, 448 mutants in the sampled mutant set, 1, 721 were killed by the test suite, resulting in a 38.7% overall mutation coverage. The difference between the overall coverage calculated from the sampled set and the one calculated from the full set is only 0.1%. Table 9.7 classifies the differences in mutation scores using the categories in Table 9.5. The first row shows the values for the weighted mutation sampling, the second row (copied from 9.6) show the values for full mutation testing. We see that the number of classes in each category remains roughly the same, thus confirming that sampling does not diminish the confidence in the validity of mutation coverage. This is confirmed in Figure 9.9, which shows the correlation between the full mutation coverage values and the sampled ones; they do indeed have a very strong correlation ($\rho = 0.99$, p-value < 0.00001).

RQ3 Summary

When a full mutation testing exceeds the time limitations imposed by a continuous integration setting (i.e. an analysis once a week during the week-end), we can use weighted mutation sampling to reduce the performance overhead and at the same time preserve an accurate estimation of the mutation coverage.

9.6 THREATS TO VALIDITY

We now identify factors that may jeopardize the validity of our results and the actions we took to reduce or alleviate the risk. Consistent with the guidelines for case studies research (see [178]) we organize them into four categories.

Internal Validity.

Threats to internal validity focus on confounding factors that can influence the obtained results. In this study, this mainly concerns equivalent mutants and a limited set of mutation operators. Because of the large number of generated mutants, it is very difficult to check for equivalent mutants in the final generated results due to the amount of manual labor needed to find and remove such mutants. Nevertheless, since equivalent mutants would add to false positives, they would be discovered when the developers are trying to create new tests or improve the available tests by referring to the information ac-

quired from mutation testing. As is common practice in today’s research, we just accept the risk [91].

Another threat stems from the limited set of mutation operators used in LittleDarwin. The addition of more mutation operators does not impact the results of our industrial case study, where mutation coverage (even with the limited set of mutation operators) produced more information than branch coverage. However, addition or removal of mutation operators can have an adverse effect on the quality of mutant sampling, since there are mutation operators that can produce a large number of redundant mutants (e.g. Null-Type mutation operators [36]), and thus affect the distribution of the sampled mutants. To identify such effects in practice, further case studies on industrial software is required; nonetheless, we did not pursue this line of research in this study.

A large majority of the produced mutants are indeed redundant, and this affects the stability of mutation coverage as a metric [72]. In order to remove the redundant mutants, mutant subsumption relationships need to be used [96]. However, determining these relationships is a very difficult task at large scale. There are no available tools to our knowledge that performs static subsumption analysis on Java programs of this scale. In addition, dynamic subsumption analysis requires a very high quality test suite to be accurate. Therefore in case of non-adequate test suites of our subject projects, the dynamic subsumption relationships are unreliable to detect redundancy among mutants [71]. For this reason, we did not filter redundant mutants in this study.

During the course of the study we discovered that JaCoCo does not report branch coverage correctly in some cases. This phenomenon has been already documented in Tengeri et al. [155, 171]. Despite this fact, we decided to use the results as is for two reasons: first, we were informed by the developers of the industrial project that no other tool was capable of integration with their environment, and second, the results of the tool were used as is for the decision making process regarding the testing of the software. Given the fact that JaCoCo is one of the most commonly used tools in maven builds, we believe its weaknesses are a reflection of the difficulties in computing branch coverage in practice, and therefore worth studying.

Construct Validity.

Threats to construct validity focus on how accurately the observations describe the phenomena of interest. This research is driven by RQ2 where we compare test coverage provided by two different tools. To minimize the risk on making wrong observations, we compare the test coverage in two different ways, once using the categories in Table 9.5 and once via the Kendall’s τ_b coefficient as described in statistic handbooks [103, 104]. Moreover, we manually inspect certain results, especially outliers.

In order to evaluate mutant sampling process, a common procedure is to create many smaller subsets of a test suite, and compare the mutation coverage obtained from sampled mutants and the mutation coverage obtained from all mutants for all of these smaller subsets of the test suite. Despite our attempts, this proved not to be possible in the industrial case. Because of the interdependencies between the tests, complicated setup process of the testing harness, and the use of shared resources, the order in which the tests are included and executed are important for a successful execution of the test suite. Therefore, it is not possible to randomly generate subsets of the original test suite. For this reason, we omitted this kind of analysis in our study.

Reliability.

Threats to reliability validity correspond to the degree to which the result depends on the tools used. The most important threat to validity concerns the tool LittleDarwin implemented by first author. Compile errors and errors in tests can affect the final results, since LittleDarwin checks only if the build process has failed or not, and it does not go further to determine the reason for the failure. This was addressed by inspecting the output of the build system. In this process, 107 invalid mutants were detected. These mutants could not be compiled, and were excluded from the final results. Another threat to reliability validity is the fact that the data gathered by JaCoCo might not be accurate (especially due to dynamic loading of components), and therefore the conclusions based on the comparison between branch coverage and mutation coverage might not be accurate enough. However, since JaCoCo is the tool that is being used in the structure of Segmentation component to acquire this information in the first place, the conclusions are still relevant by challenging the previously held beliefs about the system. Because of active development of JaCoCo and its popularity as an integral part of continuous integration systems, it is debatable whether better accuracy can be achieved.

External Validity.

Threats to external validity correspond to the generalizability of our results. Since this study was performed only on a software running on a single platform with a specific target language and a specific continuous integration environment. Therefore, the results are certainly not representative for all possible industrial systems. However, it provides an outlook on the feasibility of applying mutation testing in an industrial environment, especially concerning the challenges we faced and workarounds we performed. Nevertheless, we partially addressed the generalizability by extending the analysis regarding RQ2 on other open source cases. By comparing these results with those of the industrial case, we determined the situations in which our conclusions can be generalized.

9.7 RELATED WORK

There are several studies that assess the effectiveness of test coverage metrics. Offutt and Voas [152] use generated test cases for Fortran, and conclude mutation coverage subsumes condition coverage techniques. Li and Offutt [26] compare mutation coverage with edge-pair, all-uses, and prime path coverage. They use hand-seeded faults and manually developed test cases in their comparison, and conclude that mutation coverage is more effective in detecting faults, and requires a smaller test suite to be satisfied. Gligoric et al. [74] compare non-adequate test suites by using several coverage criteria, and conclude that branch coverage is the best predictor of mutation coverage. Gopinath et al. [29] compare coverage criteria that is available to developers on a large set of open source cases, and conclude that statement coverage, and not branch coverage, is the best predictor of mutation coverage.

Literature is clear on the dangers of using code coverage as a threshold for quality of the test suite. Marick [179] points out a scenario that relying solely on the code coverage metrics could result in faults not being detected. Inozemtseva and Holmes [76] compared decision coverage, modified condition coverage, and statement coverage on large subjects using generated test cases and mutation coverage as the effectiveness criteria, and concluded that while code coverage is good for identifying under-tested parts of the subject, it should not be used as a quality target. Aaltonen et al. [180] reach the same conclusion in their analysis comparing mutation coverage and code coverage metrics in assessment of students' skills. They propose adoption of both metrics in order to have an accurate assessment of the test suite quality. Smith and Williams [149, 150] studied the effects of using mutation testing to augment a test suite. They concluded that developing new test cases that increase the mutation coverage also increases branch coverage, and statement coverage of the test suite. They also conclude that the inclusion of new mutation operators is less important than the speed and efficiency of mutation testing process. Andrews et al. [120] validate the use of mutation testing as a benchmark for other coverage criteria using an industrial case with known faults, and conclude that not only mutation testing can be used in a research context, but also in a practical context, it can be used as a threshold to develop new test cases. Li et al. [181] use industrial cases written in Ruby and demonstrate that using mutation testing still adds value to a test suite that has 97% statement coverage.

9.8 CONCLUSION AND FUTURE WORK

With the increasing interest in continuous integration and its reliance on fully automated tests, modern software development teams continuously monitor the coverage of

the test suite as well. Unfortunately, the state of the practice is reluctant to adopt strong coverage metrics (namely mutation coverage), instead relying on weaker kinds of coverage (namely branch coverage). We argue that there are three issues for this reluctant attitude towards mutation testing: (a) the complexity of continuous build environments and (b) the perception that branch coverage is “good enough”; (c) the performance overhead during the build. Consequently, we set out to investigate the pros and cons that arise when adopting mutation testing in an industrial continuous integration setting, namely the Segmentation component of the Impax ES medical imaging software used by Agfa HealthCare. The Impax ES system is configured as a product-line released in two main variants (production or prototype), with a few minor variants for the target hardware platform. The extensive use of (dynamic) OSGI headers implies a complicated build process where the Maven plug-in Tycho [<https://eclipse.org/tycho/>] is used to fetch dependencies, compile source files, and run the test suite. This lead us to pursue the following research questions.

RQ1: *Is it feasible to integrate mutation testing in a continuous integration system?*

- Byte-level mutation tools such as PITest cannot be easily integrated into a complicated build process. Yet, if one decouples the mutation tool from the test infrastructure (thus relies on the build system to manipulate the tests) it is feasible to integrate mutation testing in a continuous integration setting. However, one does so at the expense of performance — mutation testing cannot be done as smartly and efficiently as the tightly coupled counterpart.

RQ2: *Does mutation testing reveal additional weaknesses in the test suite compared to branch coverage?*

- For all the cases under investigation, we discovered that mutation tests reveals additional weaknesses compared to branch coverage. More specifically, there were several classes which had a high branch coverage yet a low mutation coverage, hence in such situations branch coverage gives a false sense of confidence. However, the added value of the mutation tests is context dependent and varies quite a lot between the cases we investigated. This warrants further research into the nature of weak coverage values both for branch coverage and mutation coverage. Especially because here as well, we noticed that dynamic loading of components interferes with calculating branch coverage.

RQ3: *Can we reduce the performance overhead induced by mutation testing to an acceptable level?*

- When a full mutation testing exceeds the time limitations imposed by a continuous integration setting (i.e. an analysis once a week during the week-end), we can reduce the performance overhead by means of weighted mutation sampling without sacrificing the fault detection capability.

The contributions of this research are fourfold. First, we report the challenges that arise when mutation testing is integrated in a continuous integration tool of a real industrial case. Second, we adapt and use our mutation testing tool called LittleDarwin to overcome these challenges. Third, we perform a joint analysis of mutation and branch coverage on cases with non-adequate test suites. For this analysis we used four open source cases (totaling close to 40 thousand lines of code with varying degrees of test coverage) and one industrial case (with more than 38 thousand of lines of code with limited unit test coverage). We demonstrate that in cases without full branch test adequacy, mutation testing does not subsume (as expected) branch coverage. Yet, it provides complementary information that can be exploited for both determining the fault detection ability of the test suite, and forming a long-term plan to improve it. We also did not find sufficient evidence to support previous conclusions in literature regarding the relation of branch coverage and mutation coverage, namely, we cannot confirm that branch coverage is a good estimator of mutation coverage in complicated systems. Fourth, we describe how to adapt mutation testing in order to satisfy industrial time constraints and yet preserve its ability to evaluate the quality of test suite.

Future Work. There are several ideas following this study that are worthy of further investigation. In this work, we investigate whether mutation testing reveals more weaknesses in the test suite compared to branch coverage. Given the fact that the ultimate goal of software testing is to reveal as many faults as possible, it is interesting to investigate whether test suites optimized for mutation coverage are more successful in finding faults when compared to test suites optimized for branch coverage. Similarly, the validity of mutant sampling can be investigated deeper by comparing the fault detection capability of test suites optimized for sampled and full-set of mutants. In addition, including more mutation operators to increase the density of mutants in code might increase the efficiency of random sampling. Another prospective research topic is to replicate the results of this study using a minimal set of mutants by detecting the redundant mutants through subsumption analysis. The feasibility of performing subsumption analysis on large industrial software is still in question, and therefore, worth further study.

Conclusion

Software faults cost more than 1.7 trillion USD yearly. Yet, in the current competitive market, the focus of industry is to deliver products as fast as possible, and there is only a limited amount of resources allocated to testing software systems. While no amount of software testing guarantees absence of faults, using methods and metrics that are developed and studied in academia allows industry to improve their test quality, and in turn, reduce the impact of software faults.

One of the methods that is advocated by academia to assess the quality of software tests is mutation testing. Extensive research over past 4 decades has demonstrated that mutation testing is superior to other methods and metrics when it comes to the assessment of test quality. Yet it is not yet adopted industry-wide.

In this thesis we attempted to tackle this problem by identifying the industrial needs, and removing some of the obstacles to the adoption of mutation testing through industrial case studies in direct or indirect collaboration with industrial partners. We identified three main problems:

- **Performance Problem:** Mutation testing is computationally intensive.
- **Fault Model Problem:** Mutation testing requires a fault model that represents the common faults in its target context.
- **Tool Problem:** Mutation testing tools cannot handle the complexity of industrial software.

For the performance problem, we evaluated the use of techniques such as mutant sampling, higher-order mutation, dynamic subsumption analysis, and the use of change-based mutation testing in the field. In each case, we described the trade-offs of using these techniques, and identified their strong and weak points.

For the fault model problem, we proposed the use of new mutation operators to target null-type faults in Java based on feedback from our industrial partners, and C++11/14 features in C++ based on common faults described by domain experts.

For the tool problem, we created a mutation testing tool called LittleDarwin to handle the complexities associated with the industrial software, including the build and testing structure and integration with the continuous integration pipeline. We then used our tool in an industrial case study on a real life safety critical software and compared its results to the common tools and methods used in industry. In addition, we performed change-based mutation testing in collaboration with an industrial partner on real life industrial systems.

While there remains many obstacles on the wide-spread industrial adoption of mutation testing, the amount of interest we received from our industrial partners and the attention that is given to mutation testing makes us to believe that we have already taken the first steps towards industrial adoption of mutation testing.

Bibliography

- [1] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*, ICSE '06, pages 827–830, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-375-1. doi: 10.1145/1134285.1134425. URL <http://dx.doi.org/10.1145/1134285.1134425>. (Cited on pages xviii, 8, 19, 21, and 119).
- [2] Michael Newman. Software errors cost u.s. economy 59.5 billion usd annually, 2002. URL http://www.abeacha.com/NIST_press_release_bugs_cost.htm. (Cited on page 1).
- [3] Wolfgang Platz. Software fails watch, 5th edition, 2019. URL <https://www.tricentis.com/wp-content/uploads/2019/01/Software-Fails-Watch-5th-edition.pdf>. (Cited on page 1).
- [4] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, January 2001. ISSN 0018-9162. doi: 10.1109/2.962984. URL <http://dx.doi.org/10.1109/2.962984>. (Cited on page 1).
- [5] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, mar 2015. ISSN 0164-1212. doi: 10.1016/j.jss.2014.12.027. URL <http://www.sciencedirect.com/science/article/pii/S0164121214002854>. (Cited on page 1).
- [6] John D. McGregor. Test early, test often. *Journal of Object Technology*, 6(4):7–14, May 2007. ISSN 1660-1769. doi: 10.5381/jot.2007.6.4.c1. URL <http://dx.doi.org/10.5381/jot.2007.6.4.c1>. (column). (Cited on pages 1, 12, and 50).
- [7] Vahid Garousi and Junji Zhi. A survey of software testing practices in Canada. *Journal of Systems and Software*, 86(5):1354–1376, may 2013. ISSN 0164-1212. doi: 10.1016/j.jss.2012.12.051. URL <http://www.sciencedirect.com/science/article/pii/S0164121212003561>. (Cited on pages 1, 2, 32, and 57).

BIBLIOGRAPHY

- [8] Philipp Diebold and Marc Dahlem. Agile practices in practice: A mapping study. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, pages 30:1–30:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2476-2. doi: 10.1145/2601248.2601254. URL <http://doi.acm.org/10.1145/2601248.2601254>. (Cited on page 1).
- [9] Vahid Garousi, Ahmet CoÅşkunÃğay, Aysu Betin-Can, and Onur DemirÃúr. A survey of software engineering practices in turkey. *Journal of Systems and Software*, 108:148 – 177, 2015. ISSN 0164-1212. doi: <https://doi.org/10.1016/j.jss.2015.06.036>. URL <http://www.sciencedirect.com/science/article/pii/S0164121215001314>. (Cited on page 1).
- [10] Nicolli S.R. Alves, Thiago S. Mendes, Manoel G. de Mendonça, Rodrigo O. Spínola, Forrest Shull, and Carolyn Seaman. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, 70:100–121, feb 2016. ISSN 0950-5849. doi: 10.1016/j.infsof.2015.10.008. URL <http://www.sciencedirect.com/science/article/pii/S0950584915001743>. (Cited on page 1).
- [11] Vahid Garousi, Matt M. Eskandar, and Kadir Herkiloğlu. Industry–academia collaborations in software testing: experience and success stories from canada and turkey. *Software Quality Journal*, 25(4):1091–1143, Dec 2017. ISSN 1573-1367. doi: 10.1007/s11219-016-9319-5. URL <https://doi.org/10.1007/s11219-016-9319-5>. (Cited on page 1).
- [12] David Tenegeri, Arpad Beszedes, David Havas, and Tibor Gyimothy. Toolset and program repository for code coverage-based test suite analysis and manipulation. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 47–52. IEEE, sep 2014. doi: 10.1109/SCAM.2014.38. URL <http://dx.doi.org/10.1109/SCAM.2014.38>. (Cited on pages 1 and 113).
- [13] Benjamin Brosgol. Do-178C: The next avionics safety standard. *ACM SIGAda Ada Letters*, 31(3):5–6, nov 2011. ISSN 1094-3641. doi: 10.1145/2070336.2070341. URL <http://doi.acm.org/10.1145/2070336.2070341>. (Cited on pages 1 and 116).
- [14] ISO. Road vehicles – Functional safety, 2011. (Cited on pages 1 and 116).
- [15] Yi Wei, Bertrand Meyer, and Manuel Oriol. Is branch coverage a good measure of testing effectiveness? In Bertrand Meyer and Martin Nordio, editors, *Empirical Software Engineering and Verification*, volume 7007 of *Lecture Notes in Computer Science*, pages 194–212. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-25230-3. doi: 10.1007/978-3-642-25231-0_5. URL http://dx.doi.org/10.1007/978-3-642-25231-0_5. (Cited on pages 2, 113, and 116).

- [16] Hadi Hemmati. How effective are code coverage criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 151–156. IEEE, aug 2015. doi: 10.1109/QRS.2015.30. (Cited on pages 2, 113, and 116).
- [17] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 597–608, May 2017. doi: 10.1109/ICSE.2017.61. (Cited on pages 2 and 95).
- [18] John Joseph Chilenski. An investigation of three forms of the modified condition decision coverage (mc/dc) criterion. Technical report, Office of Aviation Research, 2001. (Cited on page 2).
- [19] Yuen Tak Yu and Man Fai Lau. A comparison of mc/dc, mumcut and several other coverage criteria for logical decisions. *Journal of Systems and Software*, 79(5):577 – 590, 2006. ISSN 0164-1212. doi: <https://doi.org/10.1016/j.jss.2005.05.030>. URL <http://www.sciencedirect.com/science/article/pii/S0164121205001202>. Quality Software. (Cited on page 2).
- [20] M. P. E. Heimdahl, M. W. Whalen, A. Rajan, and M. Staats. On mc/dc and implementation structure: An empirical study. In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pages 5.B.3–1–5.B.3–13, Oct 2008. doi: 10.1109/DASC.2008.4702848. (Cited on page 2).
- [21] Larry Joe Morell. *A Theory of Error-based Testing*. PhD thesis, University of Maryland at College Park, College Park, MD, USA, 1983. AAI8419537. (Cited on page 2).
- [22] D. J. Richardson and M. C. Thompson. The RELAY model of error detection and its application. In *Proceedings of 1988 Second Workshop on Software Testing, Verification, and Analysis*, pages 223–230. IEEE Comput. Soc. Press, July 1988. doi: 10.1109/WST.1988.5378. (Cited on page 2).
- [23] Mike Papadakis, Marinis Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: An analysis and survey. *Advances in Computers*, 2018. ISSN 0065-2458. doi: 10.1016/bs.adcom.2018.03.015. URL <http://www.sciencedirect.com/science/article/pii/S0065245818300305>. (Cited on pages 2, 8, 80, 94, 112, 113, and 119).
- [24] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering - ICSE '05, ICSE '05*, pages 402–411, New York, NY, USA,

BIBLIOGRAPHY

2005. ACM Press. ISBN 1-58113-963-2. doi: 10.1145/1062455.1062530. (Cited on pages 2, 12, 28, 50, 70, and 112).
- [25] Phyllis G. Frankl, Stewart N. Weiss, and Cang Hu. All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, sep 1997. ISSN 0164-1212. doi: 10.1016/s0164-1212(96)00154-9. URL [http://dx.doi.org/10.1016/S0164-1212\(96\)00154-9](http://dx.doi.org/10.1016/S0164-1212(96)00154-9). (Cited on pages 2, 12, and 114).
- [26] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *2009 International Conference on Software Testing, Verification, and Validation Workshops – Mutation Workshop*, pages 220–229. IEEE, April 2009. doi: 10.1109/ICSTW.2009.30. URL <http://dx.doi.org/10.1109/ICSTW.2009.30>. (Cited on pages 2, 112, 114, 131, and 145).
- [27] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and Rene Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops – Mutation Workshop*, pages 47–53. IEEE, apr 2018. doi: 10.1109/ICSTW.2018.00027. (Cited on page 2).
- [28] Goran Petrović and Marko Ivanković. State of mutation testing at google. In *Proceedings of the 40th International Conference on Software Engineering Software Engineering in Practice - ICSE-SEIP '18, ICSE-SEIP '18*, pages 163–171, New York, NY, USA, 2018. ACM Press. ISBN 978-1-4503-5659-6. doi: 10.1145/3183519.3183521. URL <http://doi.acm.org/10.1145/3183519.3183521>. (Cited on pages 2 and 3).
- [29] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014, ICSE 2014*, pages 72–82, New York, NY, USA, 2014. ACM Press. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568278. (Cited on pages 2, 113, 116, and 145).
- [30] A. Jefferson Offutt and Roland H. Untch. Mutation 2000: Uniting the orthogonal. In W. Eric Wong, editor, *Mutation Testing for the New Century*, volume 24 of *The Springer International Series on Advances in Database Systems*, pages 34–44. Springer US, 2001. ISBN 978-1-4419-4888-5. doi: 10.1007/978-1-4757-5939-6_7. URL http://dx.doi.org/10.1007/978-1-4757-5939-6_7. (Cited on pages 2, 5, 8, 12, 28, 50, 70, 113, 117, 120, and 122).
- [31] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, sep 2011.

- ISSN 0098-5589. doi: 10.1109/tse.2010.62. URL <http://dx.doi.org/10.1109/TSE.2010.62>. (Cited on pages 2, 9, 12, 13, 14, 28, 31, 50, 67, 71, 72, 80, 94, 112, 113, and 120).
- [32] Ali Parsai, Alessandro Murgia, and Serge Demeyer. LittleDarwin: A feature-rich and extensible mutation testing framework for large and complex java systems. In Mehdi Dastani and Marjan Sirjani, editors, *Fundamentals of Software Engineering: 7th International Conference, FSEN 2017, Tehran, Iran, April 26–28, 2017, Revised Selected Papers*, pages 148–163. Springer International Publishing, Cham, 2017. ISBN 978-3-319-68972-2. doi: 10.1007/978-3-319-68972-2_10. URL https://doi.org/10.1007/978-3-319-68972-2_10. (Cited on pages 4, 71, 79, 80, and 122).
- [33] Ali Parsai, Alessandro Murgia, and Serge Demeyer. Evaluating random mutant selection at class-level in projects with non-adequate test suites. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering - EASE 16, EASE '16*, pages 11:1–11:10, New York, NY, USA, 2016. ACM Press. ISBN 978-1-4503-3691-8. doi: 10.1145/2915970.2915992. URL <http://doi.acm.org/10.1145/2915970.2915992>. (Cited on pages 4, 8, 10, 13, 21, 23, 54, 66, 70, 71, 80, 120, and 141).
- [34] Ali Parsai, Alessandro Murgia, and Serge Demeyer. A model to estimate first-order mutation coverage from higher-order mutation coverage. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 365–373. IEEE, aug 2016. doi: 10.1109/qrs.2016.48. (Cited on pages 4, 10, 13, 21, 23, 71, and 80).
- [35] Ali Parsai and Serge Demeyer. Dynamic mutant subsumption analysis using LittleDarwin. In *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing - A-TEST 2017, A-TEST 2017*, pages 1–4, New York, NY, USA, 2017. ACM Press. ISBN 978-1-4503-5155-3. doi: 10.1145/3121245.3121249. URL <http://doi.acm.org/10.1145/3121245.3121249>. (Cited on page 4).
- [36] Ali Parsai and Serge Demeyer. Do null-type mutation operators help prevent null-type faults? In Barbara Catania, Rastislav Kráľovič, Jerzy Nawrocki, and Giovanni Pighizzini, editors, *SOFSEM 2019: Theory and Practice of Computer Science*, pages 419–434, Cham, 2019. Springer International Publishing. ISBN 978-3-030-10801-4. doi: 10.1007/978-3-030-10801-4_33. (Cited on pages 4, 7, 119, and 143).
- [37] Ali Parsai, Serge Demeyer, and Sèph De Busser. C++11/14 mutation operators based on common fault patterns. In Inmaculada Medina-Bulo, Mercedes G. Merayo, and Robert Hierons, editors, *Testing Software and Systems*, pages 102–118, Cham, 2018. Springer International Publishing. ISBN 978-3-319-99927-2. doi: 10.1007/978-3-319-99927-2_9. (Cited on pages 4, 7, 81, and 119).

BIBLIOGRAPHY

- [38] Richard A. DeMillo, Richard J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, apr 1978. ISSN 0018-9162. doi: 10.1109/c-m.1978.218136. URL <http://dx.doi.org/10.1109/C-M.1978.218136>. (Cited on pages 5, 12, 13, 28, 50, 52, 70, 71, 78, and 117).
- [39] Timothy Alan Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, USA, 1980. AAI8025191. (Cited on pages 5, 8, 13, 14, 30, 47, 52, 71, 117, and 120).
- [40] Bernhard J. M. GrÃijjn, David Schuler, and Andreas Zeller. The impact of equivalent mutants. In *2009 International Conference on Software Testing, Verification, and Validation Workshops – Mutation Workshop*, pages 192–199. IEEE, April 2009. doi: 10.1109/ICSTW.2009.37. URL <http://dx.doi.org/10.1109/ICSTW.2009.37>. (Cited on pages 6, 31, 46, 52, and 117).
- [41] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering*, 40(1):23–42, jan 2014. ISSN 0098-5589. doi: 10.1109/TSE.2013.44. (Cited on pages 6, 52, and 117).
- [42] A. Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, sep 1997. doi: 10.1002/(sici)1099-1689(199709)7:3<165::aid-stvr143>3.0.co;2-u. URL [http://dx.doi.org/10.1002/\(SICI\)1099-1689\(199709\)7:3<165::AID-STVR143>3.0.CO;2-U](http://dx.doi.org/10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U). (Cited on pages 6, 14, 71, 80, 96, and 117).
- [43] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 936–946. IEEE, may 2015. doi: 10.1109/icse.2015.103. (Cited on pages 7 and 118).
- [44] A. Jefferson Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. An experimental evaluation of data flow and mutation testing. *Software: Practice and Experience*, 26(2):165–176, feb 1996. ISSN 1097-024X. doi: 10.1002/(SICI)1097-024X(199602)26:2<165::AID-SPE5>3.0.CO;2-K. URL [http://dx.doi.org/10.1002/\(SICI\)1097-024X\(199602\)26:2<165::AID-SPE5>3.0.CO;2-K](http://dx.doi.org/10.1002/(SICI)1097-024X(199602)26:2<165::AID-SPE5>3.0.CO;2-K). (Cited on pages 7 and 118).
- [45] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, dec 1997. ISSN 0360-0300. doi: 10.1145/267580.267590. URL <http://dx.doi.org/10.1145/267580.267590>. (Cited on pages 7, 34, 116, and 118).

- [46] R.A. DeMilli and A.J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, Sep 1991. ISSN 0098-5589. doi: 10.1109/32.92910. (Cited on pages 7 and 118).
- [47] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, mar 2012. ISSN 0098-5589. doi: 10.1109/TSE.2011.93. URL <http://dx.doi.org/10.1109/TSE.2011.93>. (Cited on pages 7, 10, 80, and 118).
- [48] Kim N. King and A. Jefferson Offutt. A fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, jul 1991. ISSN 1097-024X. doi: 10.1002/spe.4380210704. URL <http://dx.doi.org/10.1002/spe.4380210704>. (Cited on pages 7, 8, 13, 79, 94, 96, 119, and 120).
- [49] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, apr 1996. ISSN 1049-331X. doi: 10.1145/227607.227610. (Cited on pages 7, 13, 18, 31, 46, 54, 70, 78, 96, and 119).
- [50] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for java. In *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.*, pages 352–363. IEEE Comput. Soc, 2002. doi: 10.1109/issre.2002.1173287. URL <http://dx.doi.org/10.1109/ISSRE.2002.1173287>. (Cited on pages 7, 13, 78, 80, and 119).
- [51] Huo Yan Chen and Su Hu. Two new kinds of class level mutants for object-oriented programs. In *2006 IEEE International Conference on Systems, Man and Cybernetics*, volume 3, pages 2173–2178. IEEE, oct 2006. doi: 10.1109/icsmc.2006.385183. URL <http://dx.doi.org/10.1109/ICSMC.2006.385183>. (Cited on pages 7 and 119).
- [52] Hyo-Jeong Lee, Yu-Seong Ma, and Yong-Rae Kwon. Empirical evaluation of orthogonality of class mutation operators. In *11th Asia-Pacific Software Engineering Conference, APSEC '04*, pages 512–518, Washington, DC, USA, 2004. IEEE. ISBN 0-7695-2245-9. doi: 10.1109/apsec.2004.49. URL <http://dx.doi.org/10.1109/APSEC.2004.49>. (Cited on pages 7 and 119).
- [53] Anna Derezińska and Marcin Rudnik. Quality evaluation of object-oriented and standard mutation operators applied to c# programs. In Carlo A. Furia and Sebastian Nanz, editors, *Objects, Models, Components, Patterns*, pages 42–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-30561-0. doi: 10.1007/978-3-642-30561-0_5. URL http://dx.doi.org/10.1007/978-3-642-30561-0_5. (Cited on pages 7 and 119).

BIBLIOGRAPHY

- [54] Zaheed Ahmed, Muhammad Zahoor, and Irfan Younas. Mutation operators for object-oriented systems: A survey. In *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, volume 2, pages 614–618. IEEE, feb 2010. doi: 10.1109/ICCAE.2010.5451692. URL <http://dx.doi.org/10.1109/ICCAE.2010.5451692>. (Cited on pages 7 and 119).
- [55] Hossain Shahriar and Mohammad Zulkernine. Mutation-based testing of format string bugs. In *2008 11th IEEE High Assurance Systems Engineering Symposium, HASE '08*, pages 229–238, Washington, DC, USA, dec 2008. IEEE. ISBN 978-0-7695-3482-4. doi: 10.1109/hase.2008.8. URL <http://dx.doi.org/10.1109/HASE.2008.8>. (Cited on pages 7 and 119).
- [56] Fanping Zeng, Liangliang Mao, Zhide Chen, and Qing Cao. Mutation-based testing of integer overflow vulnerabilities. In *2009 5th International Conference on Wireless Communications, Networking and Mobile Computing, WiCOM'09*, pages 4416–4419, Piscataway, NJ, USA, sep 2009. IEEE. ISBN 978-1-4244-3692-7. doi: 10.1109/wicom.2009.5302048. URL <http://dx.doi.org/10.1109/WICOM.2009.5302048>. (Cited on pages 7 and 119).
- [57] Robin Abraham and Martin Erwig. Mutation operators for spreadsheets. *IEEE Transactions on Software Engineering*, 35(1):94–108, jan 2009. ISSN 0098-5589. doi: 10.1109/TSE.2008.73. URL <http://dx.doi.org/10.1109/TSE.2008.73>. (Cited on pages 7 and 119).
- [58] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. ExMAN: A generic and customizable framework for experimental mutation analysis. In *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, MUTATION '06, page 4, Washington, DC, USA, nov 2006. IEEE. ISBN 0-7695-2897-X. doi: 10.1109/mutation.2006.5. URL <http://dx.doi.org/10.1109/MUTATION.2006.5>. (Cited on pages 7 and 119).
- [59] Rodolfo Adamshuk Silva, Simone do Rocio Senger de Souza, and Paulo Sergio Lopes de Souza. Mutation operators for concurrent programs in MPI. In *2012 13th Latin American Test Workshop (LATW), LATW '12*, pages 1–6, Washington, DC, USA, apr 2012. IEEE. ISBN 978-1-4673-2355-0. doi: 10.1109/latw.2012.6261240. URL <http://dx.doi.org/10.1109/LATW.2012.6261240>. (Cited on pages 7 and 119).
- [60] Allen Troy Acree Jr. *On Mutation*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1980. (Cited on pages 8, 14, 30, 47, and 120).
- [61] Weichen Eric Wong. *On Mutation and Data Flow*. PhD thesis, Purdue University, West Lafayette, IN, USA, 1993. UMI Order No. GAX94-20921. (Cited on pages 8, 28, 30, 33, 38, and 120).

- [62] Aditya P. Mathur and W. Eric Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability*, 4(1):9–31, 1994. ISSN 1099-1689. doi: 10.1002/stvr.4370040104. (Cited on pages 8 and 120).
- [63] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. Is operator-based mutant selection superior to random mutant selection? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, ICSE '10, pages 435–444, New York, NY, USA, 2010. ACM Press. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806863. URL <http://doi.acm.org/10.1145/1806799.1806863>. (Cited on pages 8, 28, 29, 30, 31, 33, 38, 47, 70, 120, and 139).
- [64] W.Eric Wong and Aditya P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, dec 1995. ISSN 0164-1212. doi: 10.1016/0164-1212(94)00098-0. URL <http://www.sciencedirect.com/science/article/pii/0164121294000980>. (Cited on pages 8, 28, 30, 31, 38, 46, 47, 70, and 120).
- [65] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. Operator-based and random mutant selection: Better together. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 92–102. IEEE, nov 2013. doi: 10.1109/ASE.2013.6693070. URL <http://dx.doi.org/10.1109/ASE.2013.6693070>. (Cited on pages 8, 12, 28, 29, 30, 31, 33, 36, 47, 70, and 120).
- [66] Mehmet Sahinoglu and Eugene H. Spafford. A bayes sequential statistical procedure for approving software products. In *Proceedings of the IFIP Conference on Approving Software Products (ASP '90)*, pages 43–56, 1990. (Cited on pages 9, 31, and 120).
- [67] Yue Jia and Mark Harman. Constructing subtle faults using higher order mutation testing. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 249–258. IEEE, sep 2008. ISBN 978-0-7695-3353-7. doi: 10.1109/scam.2008.36. URL <http://ieeexplore.ieee.org/abstract/document/4637557>/<http://ieeexplore.ieee.org/document/4637557/>. (Cited on pages 9, 14, 50, 53, and 72).
- [68] Yue Jia and Mark Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379–1393, oct 2009. ISSN 0950-5849. doi: 10.1016/j.infsof.2009.04.016. URL <http://www.sciencedirect.com/science/article/pii/S0950584909000688>. (Cited on pages 9, 12, 50, 51, 53, and 67).
- [69] Bob Kurtz, Paul Ammann, and Jeff Offutt. Static analysis of mutant subsumption. In *2015 IEEE Eighth International Conference on Software Testing, Verification*

BIBLIOGRAPHY

- and Validation Workshops – Mutation Workshop*, pages 1–10. IEEE, apr 2015. doi: 10.1109/icstw.2015.7107454. (Cited on pages 9, 14, 22, 72, and 80).
- [70] D. Richard Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering and Methodology*, 8(4):411–424, oct 1999. ISSN 1049-331X. doi: 10.1145/322993.322996. (Cited on pages 9, 14, 72, and 80).
- [71] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 21–30. IEEE, mar 2014. doi: 10.1109/icst.2014.13. (Cited on pages 9, 12, 14, 67, 70, 72, 80, and 143).
- [72] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, ISSTA 2016, pages 354–365, New York, NY, USA, 2016. ACM Press. ISBN 978-1-4503-4390-9. doi: 10.1145/2931037.2931040. (Cited on pages 9, 12, 14, 70, 72, 75, 80, and 143).
- [73] David Schuler and Andreas Zeller. Javalanche: efficient mutation testing for java. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - ESEC/FSE 2009, ESEC/FSE '09*, pages 297–298, New York, NY, USA, 2009. ACM Press. ISBN 978-1-60558-001-2. doi: 10.1145/1595696.1595750. (Cited on pages 10, 19, 20, 21, 79, and 80).
- [74] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*, ISSTA 2013, pages 302–313, New York, NY, USA, 2013. ACM Press. ISBN 978-1-4503-2159-4. doi: 10.1145/2483760.2483769. (Cited on pages 10, 46, 80, 113, 114, 128, 131, and 145).
- [75] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: a practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, ISSTA 2016, pages 449–452, New York, NY, USA, 2016. ACM Press. ISBN 978-1-4503-4390-9. doi: 10.1145/2931037.2948707. (Cited on pages 10, 19, 79, and 80).
- [76] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, ICSE 2014, pages 435–445, New York, NY, USA, 2014.

- ACM Press. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568271. URL <http://doi.acm.org/10.1145/2568225.2568271>. (Cited on pages 10, 80, 114, 116, 121, and 145).
- [77] Ali Parsai, Quinten David Soetens, Alessandro Murgia, and Serge Demeyer. Considering polymorphism in change-based test suite reduction. In Torgeir Dingsøy, Nils Brede Moe, Roberto Tonelli, Steve Counsell, Cigdem Gencel, and Kai Petersen, editors, *Lecture Notes in Business Information Processing*, pages 166–181. Springer International Publishing, Cham, 2014. ISBN 978-3-319-14358-3. doi: 10.1007/978-3-319-14358-3_14. URL http://dx.doi.org/10.1007/978-3-319-14358-3_14. (Cited on pages 10, 21, 80, and 121).
- [78] Ali Parsai, Alessandro Murgia, Quinten David Soetens, and Serge Demeyer. Mutation testing as a safety net for test code refactoring. In *Scientific Workshop Proceedings of the XP2015 on - XP 2015 workshops*, XP '15 workshops, pages 8:1–8:7, New York, NY, USA, 2015. ACM Press. ISBN 978-1-4503-3409-9. doi: 10.1145/2764979.2764987. (Cited on pages 10, 31, 41, 54, 66, 71, and 80).
- [79] Kent Beck. *Test-driven Development: By Example*. Kent Beck signature book. Addison-Wesley, 2003. ISBN 9780321146533. (Cited on page 12).
- [80] Martin Fowler and Matthew Foemmel. Continuous integration. Technical report, Thoughtworks, 2006. (Cited on pages 12 and 50).
- [81] René Just. The Major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, ISSTA 2014, pages 433–436, New York, NY, USA, 2014. ACM Press. ISBN 978-1-4503-2645-2. doi: 10.1145/2610384.2628053. (Cited on pages 12, 28, 50, 70, 78, and 94).
- [82] Patrick Joseph Walsh. *A Measure of Test Case Completeness*. PhD thesis, State University of New York at Binghamton, Binghamton, NY, USA, 1985. (Cited on page 12).
- [83] Mike Papadakis and Nicos Malevris. An empirical evaluation of the first and second order mutation testing strategies. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops – Mutation Workshop*, ICSTW '10, pages 90–99, Washington, DC, USA, apr 2010. IEEE Computer Society. ISBN 978-0-7695-4050-4. doi: 10.1109/ICSTW.2010.50. URL <http://dx.doi.org/10.1109/ICSTW.2010.50>. (Cited on pages 12 and 67).
- [84] Marinos Kintis, Mike Papadakis, and Nicos Malevris. Isolating first order equivalent mutants via second order mutation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 701–710. Institute of Elec-

BIBLIOGRAPHY

- trical & Electronics Engineers (IEEE), apr 2012. doi: 10.1109/ICST.2012.160. URL <http://dx.doi.org/10.1109/ICST.2012.160>. (Cited on pages 12 and 50).
- [85] Elmahdi Omar, Sudipto Ghosh, and Darrell Whitley. HOMAJ: A tool for higher order mutation testing in AspectJ and Java. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops – Mutation Workshop, ICSTW '14*, pages 165–170, Washington, DC, USA, mar 2014. IEEE Computer Society. ISBN 978-1-4799-5790-3. doi: 10.1109/ICSTW.2014.19. URL <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6825654&contentType=Conference+Publications>. (Cited on page 12).
- [86] Bob Kurtz. On the utility of dominator mutants for mutation testing. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, FSE 2016, pages 1088–1090, New York, NY, USA, 2016. Association for Computing Machinery (ACM). ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2983950. URL <http://dx.doi.org/10.1145/2950290.2983950>. (Cited on page 12).
- [87] Milos Gligoric, Lingming Zhang, Cristiano Pereira, and Gilles Pokam. Selective mutation testing for concurrent code. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*, ISSTA 2013, pages 224–234, New York, NY, USA, 2013. ACM Press. ISBN 978-1-4503-2159-4. doi: 10.1145/2483760.2483773. (Cited on page 12).
- [88] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, Alex Groce, et al. An empirical comparison of mutant selection approaches. Technical report, Oregon State University, 2015. URL <http://hdl.handle.net/1957/55691>. (Cited on page 12).
- [89] Ali Parsai. Mutation analysis: An industrial experiment. Master’s thesis, University of Antwerp, 2015. (Cited on pages 13, 22, 31, 41, 54, 66, and 71).
- [90] Sunwoo Kim, John A. Clark, and John A. McDermid. Class mutation: Mutation testing for object-oriented programs. In *Proceedings of Net Object Days 2000*, pages 9–12, 2000. (Cited on pages 13, 94, and 96).
- [91] Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8): 861–874, jun 2006. ISSN 0167-8655. doi: 10.1016/j.patrec.2005.10.010. URL <http://dx.doi.org/10.1016/j.patrec.2005.10.010>. ROC Analysis in Pattern Recognition. (Cited on pages 14, 46, and 143).
- [92] Haidar Osman, Mircea Lungu, and Oscar Nierstrasz. Mining frequent bug-fix code changes. In *2014 Software Evolution Week - IEEE Conference on Software Mainte-*

- nance, *Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 343–347. IEEE, feb 2014. doi: 10.1109/csmr-wcre.2014.6747191. (Cited on pages 18, 78, and 81).
- [93] Pitest. <http://pitest.org/>. (Cited on page 20).
- [94] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, June 2005. ISSN 0960-0833. doi: 10.1002/stvr.308. URL <http://dx.doi.org/10.1002/stvr.308>. (Cited on page 20).
- [95] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, October 2000. ISSN 0018-9162. doi: 10.1109/2.876288. URL <http://dx.doi.org/10.1109/2.876288>. (Cited on page 20).
- [96] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng. Mutant subsumption graphs. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops – Mutation Workshop*, pages 176–185, March 2014. doi: 10.1109/ICSTW.2014.20. (Cited on pages 22 and 143).
- [97] David Schuler and Andreas Zeller. (un-)covering equivalent mutants. In *2010 Third International Conference on Software Testing, Verification and Validation, ICST '10*, pages 45–54, Washington, DC, USA, 2010. Saarland Univ., Saarbrucken, Germany, IEEE Computer Society. ISBN 978-1-4244-6435-7. doi: 10.1109/ICST.2010.30. URL <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=5477100&contentType=Conference+Publications>. (Cited on page 23).
- [98] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, jul 1977. ISSN 0098-5589. doi: 10.1109/TSE.1977.231145. URL <http://dx.doi.org/10.1109/TSE.1977.231145>. (Cited on pages 28 and 50).
- [99] Q. Yang, J. J. Li, and D. M. Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, May 2007. doi: 10.1093/comjnl/bxm021. URL <http://dx.doi.org/10.1093/comjnl/bxm021>. (Cited on pages 29, 34, and 116).
- [100] Victor R. Basili. Applying the goal/question/metric paradigm in the experience factory. In *Software Quality Assurance: A Worldwide Perspective*, 1993. (Cited on pages 29 and 51).
- [101] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *Proceedings of the 15th International Conference on Software Engineering, ICSE '93*, pages 100–107, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press. ISBN 0-89791-588-7. URL <http://dl.acm.org/citation.cfm?id=257572.257597>. (Cited on pages 33, 46, and 70).

BIBLIOGRAPHY

- [102] Anil Kumar Gayen. The frequency distribution of the product-moment correlation coefficient in random samples of any size drawn from non-normal universes. *Biometrika*, 38(1-2):219–247, 1951. doi: 10.1093/biomet/38.1-2.219. URL <http://dx.doi.org/10.1093/biomet/38.1-2.219>. (Cited on page 33).
- [103] Hervé Abdi. Kendall rank correlation coefficient. In *The Concise Encyclopedia of Statistics*, pages 278–281. Springer New York, 2007. doi: 10.1007/978-0-387-32833-1_211. URL http://dx.doi.org/10.1007/978-0-387-32833-1_211. (Cited on pages 33, 128, and 143).
- [104] Alan Agresti. *Analysis of Ordinal Categorical Data*, volume 656. John Wiley & Sons, Inc., mar 2010. doi: 10.1002/9780470594001. URL <http://dx.doi.org/10.1002/9780470594001>. (Cited on pages 33, 128, and 143).
- [105] Robert K. Yin. *Case Study Research: Design and Methods*. Applied Social Research Methods. SAGE Publications, 2003. ISBN 9780761925521. (Cited on pages 41, 66, and 90).
- [106] Aditya P. Mathur. Performance, effectiveness, and reliability issues in software testing. In [1991] *Proceedings The Fifteenth Annual International Computer Software & Applications Conference*, pages 604–605. Institute of Electrical & Electronics Engineers (IEEE), September 1991. doi: 10.1109/cmepsac.1991.170248. URL <http://dx.doi.org/10.1109/CMPSAC.1991.170248>. (Cited on pages 46 and 70).
- [107] Ellen Francine Barbosa, Jose Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. Toward the determination of sufficient mutant operators for c. *Software Testing, Verification and Reliability*, 11(2):113–136, 2001. doi: 10.1002/stvr.226. URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.226>. (Cited on page 46).
- [108] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 13th international conference on Software engineering - ICSE '08, ICSE '08*, pages 351–360, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368136. (Cited on page 46).
- [109] José Carlos Maldonado, Márcio Eduardo Delamaro, Sandra C. P. F. Fabbri, Adenilson Silva Simão, Tatiana Sugeta, Auri Marcelo Rizzo Vincenzi, and Paulo Cesar Masiero. Proteum: A family of tools to support specification and program testing based on mutation. In W. Eric Wong, editor, *Mutation Testing for the New Century*, volume 24 of *The Springer International Series on Advances in Database Systems*, pages 113–116. Springer US, 2001. ISBN 978-1-4419-4888-5. doi: 10.1007/978-1-4757-5939-6_19. URL http://dx.doi.org/10.1007/978-1-4757-5939-6_19. (Cited on page 46).

- [110] Allen T. Acree, Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Mutation analysis. Technical report, DTIC Document, 1979. (Cited on page 47).
- [111] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0321146530. (Cited on page 50).
- [112] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, January 1992. ISSN 1049-331X. doi: 10.1145/125489.125473. URL <http://dx.doi.org/10.1145/125489.125473>. (Cited on pages 50, 65, 67, and 68).
- [113] Quang Vu Nguyen and Lech Madeyski. *Advanced Computational Methods for Knowledge Engineering: Proceedings of the 2nd International Conference on Computer Science, Applied Mathematics and Applications (ICCSAMA 2014)*, chapter Problems of Mutation Testing and Higher Order Mutation Testing, pages 157–172. Springer International Publishing, Cham, 2014. ISBN 978-3-319-06569-4. doi: 10.1007/978-3-319-06569-4_12. URL http://dx.doi.org/10.1007/978-3-319-06569-4_12. (Cited on page 50).
- [114] Brian Everitt. *The Cambridge Dictionary of Statistics*. Cambridge University Press (CUP), Cambridge [u.a.], 2. ed. edition, 2002. doi: 10.1017/cbo9780511779633. URL <http://dx.doi.org/10.1017/cbo9780511779633>. (Cited on page 60).
- [115] K.S. How Tai Wah. An analysis of the coupling effect i: single test data. *Science of Computer Programming*, 48(2-3):119–161, aug 2003. ISSN 0167-6423. doi: 10.1016/s0167-6423(03)00022-4. URL [http://dx.doi.org/10.1016/S0167-6423\(03\)00022-4](http://dx.doi.org/10.1016/S0167-6423(03)00022-4). (Cited on pages 65 and 68).
- [116] Macario Polo, Mario Piattini, and Ignacio García-Rodríguez. Decreasing the cost of mutation testing with second-order mutants. *Software Testing, Verification and Reliability*, 19(2):111–131, jun 2009. ISSN 0960-0833. doi: 10.1002/stvr.392. URL <http://dx.doi.org/10.1002/stvr.392>. (Cited on page 67).
- [117] M. Kintis, M. Papadakis, and N. Malevris. Evaluating mutation testing alternatives: A collateral experiment. In *2010 Asia Pacific Software Engineering Conference*, pages 300–309, Nov 2010. doi: 10.1109/APSEC.2010.42. (Cited on page 67).
- [118] Andrew Jefferson Offutt. The coupling effect: Fact or fiction. *ACM SIGSOFT Software Engineering Notes*, 14(8):131–140, November 1989. ISSN 0163-5948. doi: 10.1145/75309.75324. (Cited on page 68).
- [119] K. S. How Tai Wah. A theoretical study of fault coupling. *Software Testing, Verification and Reliability*, 10(1):3–45, mar 2000. ISSN 1099-1689. doi: 10.1002/(SICI)1099-

BIBLIOGRAPHY

- 1689(200003)10:1<3::AID-STVR196>3.0.CO;2-P. URL [http://dx.doi.org/10.1002/\(SICI\)1099-1689\(200003\)10:1<3::AID-STVR196>3.0.CO;2-P](http://dx.doi.org/10.1002/(SICI)1099-1689(200003)10:1<3::AID-STVR196>3.0.CO;2-P). (Cited on page 68).
- [120] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, aug 2006. ISSN 0098-5589. doi: 10.1109/tse.2006.83. URL <http://dx.doi.org/10.1109/TSE.2006.83>. (Cited on pages 70, 112, and 145).
- [121] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification (Java SE 8 edition)*. Oracle, java se 8 edition, 2014. (Cited on page 78).
- [122] Haidar Osman, Manuel Leuenberger, Mircea Lungu, and Oscar Nierstrasz. Tracking null checks in open-source java systems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 304–313. IEEE, mar 2016. doi: 10.1109/saner.2016.57. (Cited on pages 78, 79, and 91).
- [123] Shuhei Kimura, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Does return null matter? In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 244–253. IEEE, feb 2014. doi: 10.1109/csmr-wcre.2014.6747176. (Cited on page 78).
- [124] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. Mutation operators for concurrent java (J2SE 5.0). In *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, MUTATION '06, pages 11–11, Washington, DC, USA, nov 2006. Sch. of Comput., Queen's Univ., Kingston, ON, IEEE. ISBN 0-7695-2897-X. doi: 10.1109/mutation.2006.10. URL <https://ieeexplore.ieee.org/document/4144730>. (Cited on pages 81 and 95).
- [125] F. C. Ferrari, J. C. Maldonado, and A. Rashid. Mutation testing for Aspect-Oriented programs. In *2008 1st International Conference on Software Testing, Verification, and Validation, ICST '08*, pages 52–61, Washington, DC, USA, April 2008. Dept. of Comput. Syst., Sao Paulo Univ., Sao Carlos, IEEE. ISBN 978-0-7695-3127-4. doi: 10.1109/ICST.2008.37. (Cited on pages 81 and 95).
- [126] Rafael A.P. Oliveira, Emil Alegroth, Zebao Gao, and Atif Memon. Definition and evaluation of mutation operators for GUI-level mutation analysis. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops – Mutation Workshop*, pages 1–10. IEEE, apr 2015. doi: 10.1109/icstw.2015.7107457. (Cited on pages 81 and 95).

- [127] Lin Deng, Jeff Offutt, Paul Ammann, and Nariman Mirzaei. Mutation operators for testing android apps. *Information and Software Technology*, 81:154–168, jan 2017. ISSN 0950-5849. doi: <https://doi.org/10.1016/j.infsof.2016.04.012>. URL <http://www.sciencedirect.com/science/article/pii/S0950584916300684>. (Cited on pages 81 and 95).
- [128] Jay Nanavati, Fan Wu, Mark Harman, Yue Jia, and Jens Krinke. Mutation testing of memory-related operators. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops – Mutation Workshop*, pages 1–10. IEEE, apr 2015. doi: 10.1109/icstw.2015.7107449. (Cited on page 81).
- [129] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999. ISBN 0-201-48567-2. (Cited on page 81).
- [130] Ian Alexander. Misuse cases: use cases with hostile intent. *IEEE Software*, 20(1):58–66, jan 2003. ISSN 0740-7459. doi: 10.1109/ms.2003.1159030. (Cited on page 86).
- [131] Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, and Benoit Baudry. A comprehensive study of pseudo-tested methods. *Empirical Software Engineering*, 24(3):1195–1225, Jun 2019. ISSN 1573-7616. doi: 10.1007/s10664-018-9653-2. URL <https://doi.org/10.1007/s10664-018-9653-2>. (Cited on page 89).
- [132] Ali Parsai. Replication package. <http://parsai.net/files/research/SofSemReplicationPackage.7z>. (Cited on page 90).
- [133] R. Baker and I. Habli. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Transactions on Software Engineering*, 39(6):787–805, June 2013. ISSN 0098-5589. doi: 10.1109/TSE.2012.56. (Cited on page 94).
- [134] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Juan José Domínguez-Jiménez, Antonio García-Domínguez, and Francisco Palomo-Lozano. Class mutation operators for c++ object-oriented systems. *annals of telecommunications - annales des télécommunications*, 70(3):137–148, Apr 2015. ISSN 1958-9395. doi: 10.1007/s12243-014-0445-4. URL <https://doi.org/10.1007/s12243-014-0445-4>. (Cited on pages 94, 100, and 110).
- [135] Bjarne Stroustrup. *Programming: Principles and Practice Using C++ (2nd Edition)*. Addison-Wesley Professional, 2nd edition, 2014. ISBN 0321992784, 9780321992789. (Cited on page 95).
- [136] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Francisco Palomo-Lozano, Antonio García-Domínguez, and Juan José Domínguez-Jiménez. Assessment of

BIBLIOGRAPHY

- class mutation operators for c++ with the mucpp mutation system. *Information and Software Technology*, 81:169–184, jan 2017. ISSN 0950-5849. doi: 10.1016/j.infsof.2016.07.002. URL <http://www.sciencedirect.com/science/article/pii/S0950584916301161>. (Cited on pages 95, 96, and 110).
- [137] Warwick Irwin and Neville Churcher. A generated parser of c++. *NZ Journal of Computing*, 8(3):26–37, 2001. (Cited on page 96).
- [138] Markus Kusano and Chao Wang. CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pages 722–725, 2013. ISBN 9781479902156. doi: 10.1109/ASE.2013.6693142. URL <http://ieeexplore.ieee.org/abstract/document/6693142/>. (Cited on page 96).
- [139] Herb Sutter, Bjarne Stroustrup, and Gabriel Dos Reis. ISO/IEC JTC1/SC22/WG21 N4262: Wording for Forwarding References, 2014. URL <http://open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4262.pdf>. (Cited on page 98).
- [140] Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O’Reilly Media, Inc., 1st edition, 2014. ISBN 1491903996, 9781491903995. (Cited on pages 100, 103, 105, and 108).
- [141] Bjarne Stroustrup. C++ Core Guidelines, 2017. URL <http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>. (Cited on pages 100 and 103).
- [142] Stephan T. Lavavej. ISO/IEC JTC1/SC22/WG21 N3853: Range-Based For-Loops: The Next Generation, 2014. URL <http://open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3853.htm>. (Cited on pages 100 and 102).
- [143] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2 edition, dec 2016. doi: 10.1017/9781316771273. (Cited on page 112).
- [144] Nan Li and Jeff Offutt. Test oracle strategies for model-based testing. *IEEE Transactions on Software Engineering*, 43(4):372–395, apr 2017. ISSN 0098-5589. doi: 10.1109/tse.2016.2597136. (Cited on page 112).
- [145] Mark Harman, Yue Jia, and William B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE ’11, ESEC/FSE ’11*, pages 212–222, New York, NY, USA, 2011. ACM Press. ISBN 978-1-4503-0443-6. doi: 10.1145/2025113.2025144. URL <http://doi.acm.org/10.1145/2025113.2025144>. (Cited on page 112).

- [146] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 654–665, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635929. URL <http://doi.acm.org/10.1145/2635868.2635929>. (Cited on page 112).
- [147] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*, ICSE '06, pages 82–91, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-375-1. doi: 10.1145/1134285.1134299. URL <http://dx.doi.org/10.1145/1134285.1134299>. (Cited on page 112).
- [148] Hyunsook Do and Gregg Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, sep 2006. ISSN 0098-5589. doi: 10.1109/tse.2006.92. URL <http://dx.doi.org/10.1109/TSE.2006.92>. (Cited on page 112).
- [149] Ben H. Smith and Laurie Williams. Should software testers use mutation analysis to augment a test set? *Journal of Systems and Software*, 82(11):1819–1832, nov 2009. ISSN 0164-1212. doi: 10.1016/j.jss.2009.06.031. URL <http://dx.doi.org/10.1016/j.jss.2009.06.031>. (Cited on pages 112 and 145).
- [150] Ben H. Smith and Laurie Williams. Should software testers use mutation analysis to augment a test set? *Journal of Systems and Software*, 82(11):1819–1832, nov 2009. ISSN 0164-1212. doi: 10.1016/j.jss.2009.06.031. URL <http://dx.doi.org/10.1016/j.jss.2009.06.031>. (Cited on pages 112 and 145).
- [151] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993. ISSN 0098-5589. doi: 10.1109/32.238581. URL <http://dx.doi.org/10.1109/32.238581>. (Cited on page 112).
- [152] A. Jefferson Offutt and Jeffrey M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical report, George Mason University, 1996. (Cited on pages 112, 131, and 145).
- [153] Brian Marick. Experience with the cost of different coverage goals for testing. In *Proceedings of the Ninth Pacific Northwest Software Quality Conference*, pages 147–164, 1991. (Cited on page 113).
- [154] Jeff Tian. *Software Quality Engineering: Testing, Quality Assurance and Quantifiable Improvement*. John Wiley & Sons, Inc., jan 2009. doi: 10.1002/0471722324. URL <http://dx.doi.org/10.1002/0471722324>. (Cited on page 113).

BIBLIOGRAPHY

- [155] Vincent Blondeau, Anne Etien, Nicolas Anquetil, Sylvain Cresson, Pascal Croisy, and Stéphane Ducasse. Test case selection in industry: an analysis of issues related to static approaches. *Software Quality Journal*, 25(4):1203–1237, Dec 2017. ISSN 1573-1367. doi: 10.1007/s11219-016-9328-4. URL <https://doi.org/10.1007/s11219-016-9328-4>. (Cited on pages 113 and 143).
- [156] Lingchao Chen and Lingming Zhang. Speeding up mutation testing via regression test selection: An extensive study. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 58–69. IEEE, apr 2018. doi: 10.1109/icst.2018.00016. (Cited on page 113).
- [157] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming google-scale continuous testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP '17*, pages 233–242, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-2717-4. doi: 10.1109/ICSE-SEIP.2017.16. URL <https://doi.org/10.1109/ICSE-SEIP.2017.16>. (Cited on page 113).
- [158] Richard Hopkins and Kevin Jenkins. *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. IBM Press, 2008. (Cited on page 114).
- [159] Vahid Garousi and Erdem Yildirim. Introducing automated GUI testing and observing its benefits: An industrial case study in the context of law-practice management software. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops – NEXTA Workshop*, pages 138–145. IEEE, apr 2018. doi: 10.1109/icstw.2018.00042. (Cited on page 114).
- [160] Grady Booch. *Object Oriented Design: With Applications*. The Benjamin/Cummings Series in Ada and Software Engineering. Benjamin/Cummings Pub., 1991. ISBN 9780805300918. URL <http://books.google.be/books?id=w5VQAAAAAAAJ>. (Cited on page 115).
- [161] Martin Fowler and Matthew Foemmel. Continuous integration (original version). <http://http://www.martinfowler.com/>, September 2010. Accessed: April, 1st 2016. (Cited on page 115).
- [162] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008. ISBN 0521880386, 9780521880381. doi: 10.1017/cbo9780511809163. URL <http://dx.doi.org/10.1017/cbo9780511809163>. (Cited on page 116).
- [163] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In

- Proceedings of 16th International Conference on Software Engineering, ICSE '94*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-5855-X. doi: 10.1109/ICSE.1994.296778. (Cited on page 116).
- [164] K. J. Hayhurst and D. S. Veerhusen. A practical approach to modified condition/decision coverage. In *20th DASC. 20th Digital Avionics Systems Conference (Cat. No.01CH37219)*, volume 1, pages 1B2/1–1B2/10. IEEE, October 2001. doi: 10.1109/dasc.2001.963305. (Cited on page 116).
- [165] Gregory Gay, Matt Staats, Michael Whalen, and Mats P. E. Heimdahl. The risks of coverage-directed test case generation. *IEEE Transactions on Software Engineering*, 41(8):803–819, August 2015. ISSN 0098-5589. doi: 10.1109/tse.2015.2421011. (Cited on page 116).
- [166] Susanne Kandl and Sandeep Chandrashekar. Reasonability of MC/DC for safety-relevant software implemented in programming languages with short-circuit evaluation. *Computing*, 97(3):261–279, aug 2014. doi: 10.1007/s00607-014-0418-5. (Cited on page 116).
- [167] Nan Li, Xin Meng, Jeff Offutt, and Lin Deng. Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report). In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 380–389. IEEE, nov 2013. doi: 10.1109/ISSRE.2013.6698891. URL <http://dx.doi.org/10.1109/ISSRE.2013.6698891>. (Cited on page 121).
- [168] Werner Janjic and Colin Atkinson. Utilizing software reuse experience for automated test recommendation. In *2013 8th International Workshop on Automation of Software Test (AST), AST '13*, pages 100–106, Piscataway, NJ, USA, may 2013. IEEE. ISBN 978-1-4673-6161-3. doi: 10.1109/iwast.2013.6595799. URL <http://dl.acm.org/citation.cfm?id=2662413.2662436>. (Cited on page 121).
- [169] Sebastian Bauersfeld, Tanja E. J. Vos, and Kiran Lakhotia. Unit testing tool competitions – lessons learned. In E.J. Tanja Vos, Kiran Lakhotia, and Sebastian Bauersfeld, editors, *Future Internet Testing: First International Workshop, FITTEST 2013, Istanbul, Turkey, November 12, 2013, Revised Selected Papers*, pages 75–94. Springer International Publishing, Cham, 2014. ISBN 978-3-319-07785-7. doi: 10.1007/978-3-319-07785-7_5. URL http://dx.doi.org/10.1007/978-3-319-07785-7_5. (Cited on page 121).
- [170] Jeshua S. Kracht, Jacob Z. Petrovic, and Kristen R. Walcott-Justice. Empirically evaluating the quality of automatically generated and manually written test suites. In *2014 14th International Conference on Quality Software*, pages 256–265. IEEE, oct

BIBLIOGRAPHY

2014. doi: 10.1109/qsic.2014.33. URL <http://dx.doi.org/10.1109/QSIC.2014.33>. (Cited on page 121).
- [171] David Tengeri, Ferenc Horvath, Arpad Beszedes, Tamas Gergely, and Tibor Gyimothy. Negative effects of bytecode instrumentation on java source code coverage. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 225–235. IEEE, mar 2016. doi: 10.1109/saner.2016.61. URL <http://dx.doi.org/10.1109/SANER.2016.61>. (Cited on pages 121, 138, and 143).
- [172] Tomek Kaczanowski. *Practical Unit Testing with TestNG and Mockito*. Tomasz Kaczanowski, Cracow, 2012. ISBN 839348930X, 9788393489305. (Cited on page 121).
- [173] Iman Saleh and Khaled Nagi. HadoopMutator: A cloud-based mutation testing framework. In Ina Schaefer and Ioannis Stamelos, editors, *Lecture Notes in Computer Science*, pages 172–187. Springer International Publishing, Cham, 2014. ISBN 978-3-319-14130-5. doi: 10.1007/978-3-319-14130-5_13. URL http://dx.doi.org/10.1007/978-3-319-14130-5_13. (Cited on page 121).
- [174] B. Assylbekov, E. Gaspar, N. Uddin, and P. Egan. Investigating the correlation between mutation score and coverage score. In *2013 UKSim 15th International Conference on Computer Modelling and Simulation*, pages 347–352. IEEE, apr 2013. doi: 10.1109/UKSim.2013.28. URL <http://dx.doi.org/10.1109/UKSim.2013.28>. (Cited on page 121).
- [175] Linda Rising and Norman S. Janoff. The scrum software development process for small teams. *IEEE Software*, 17(4):26–32, 2000. ISSN 0740-7459. doi: 10.1109/52.854065. (Cited on page 124).
- [176] Siamak Haschemi and Stephan Weißleder. A generic approach to run mutation analysis. In Leonardo Bottaci and Gordon Fraser, editors, *Testing – Practice and Research Techniques*, volume 6303 of *Lecture Notes in Computer Science*, pages 155–164. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-15584-0. doi: 10.1007/978-3-642-15585-7_15. URL http://dx.doi.org/10.1007/978-3-642-15585-7_15. (Cited on page 130).
- [177] Frederick P. Brooks. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, apr 1987. ISSN 0018-9162. doi: 10.1109/mc.1987.1663532. URL <http://dx.doi.org/10.1109/MC.1987.1663532>. (Cited on page 130).
- [178] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–

- 164, dec 2008. doi: 10.1007/s10664-008-9102-8. URL <http://dx.doi.org/10.1007/s10664-008-9102-8>. (Cited on page 142).
- [179] Brian Marick and Testing Foundations. How to misuse code coverage. In *16th International Conference and Exposition on Testing Computer Software*, 1999. (Cited on page 145).
- [180] Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH 10, OOPSLA '10*, pages 153–160, New York, NY, USA, 2010. ACM Press. ISBN 978-1-4503-0240-1. doi: 10.1145/1869542.1869567. URL <http://dx.doi.org/10.1145/1869542.1869567>. (Cited on page 145).
- [181] Nan Li, Michael West, Anthony Escalona, and Vinicius H. S. Durelli. Mutation testing in practice using ruby. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops – Mutation Workshop*, pages 1–6. IEEE, apr 2015. doi: 10.1109/ICSTW.2015.7107453. (Cited on page 145).