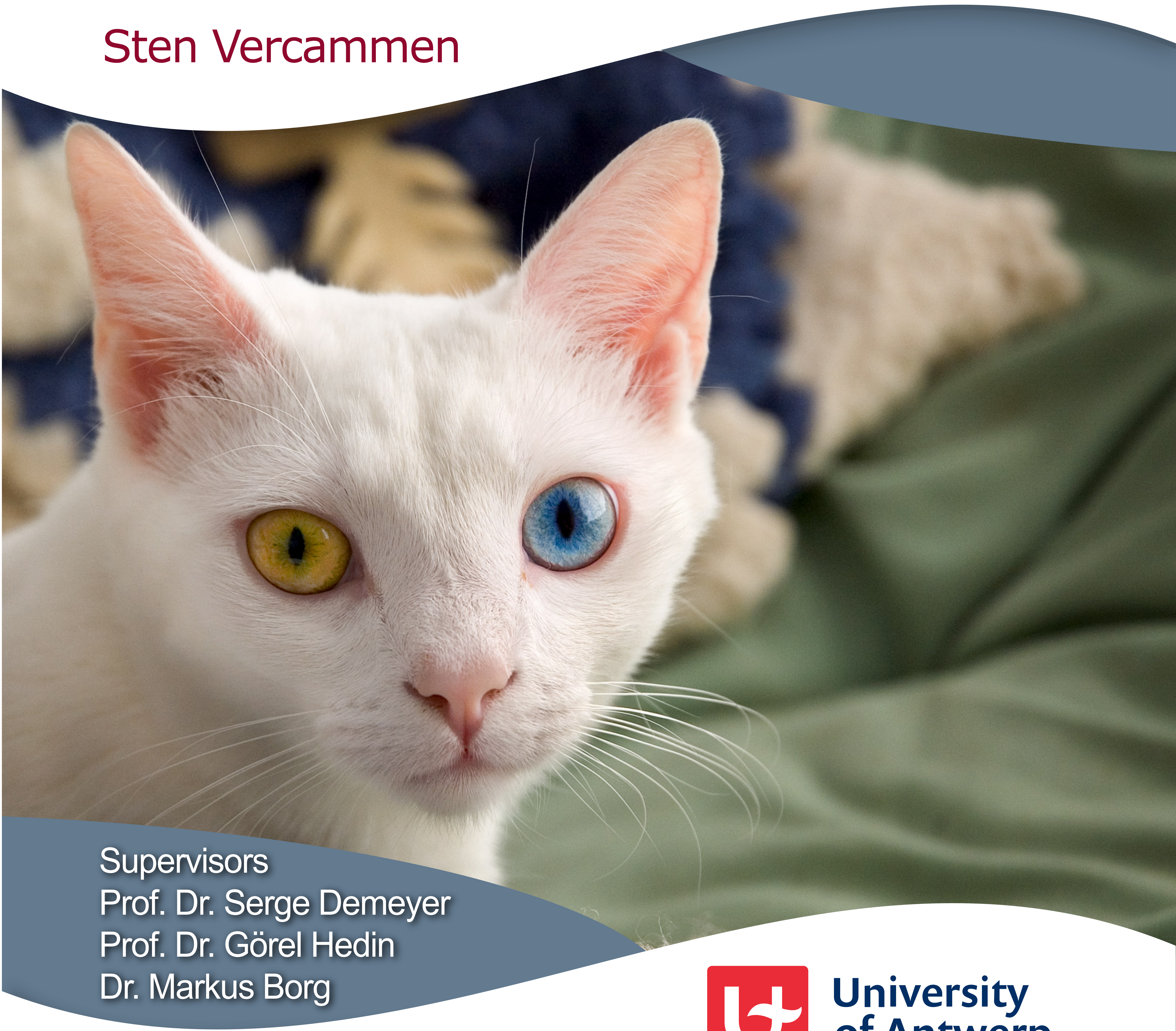
Mutation Testing: Fewer, Faster, and Smarter

Thesis submitted for the degree of:
Doctor of Science: Computer Science at the University of Antwerp, Belgium
Doctor of Philosophy in Engineering at Lund University, Sweden
to be defended by:

Sten Vercammen



Supervisors
Prof. Dr. Serge Demeyer
Prof. Dr. Görel Hedin
Dr. Markus Borg

University of Antwerp

Faculty of Science, Department of Computer Science
Antwerp, Belgium, 2023

LUND UNIVERSITY

Faculty of Engineering, Department of Computer Science
Lund, Sweden, 2023

This thesis is submitted:
- at the Faculty of Science at the University of Antwerp, in partial fulfilment of the requirements for the degree of Doctor of Science: Computer Science.
- to the Research Education Board of the Faculty of Engineering at Lund University, in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Engineering.

LU-CS-DISS: 2023-02
Dissertation 71, 2023

ISBN 978-91-8039-576-2 (printed version)
ISBN 978-91-8039-577-9 (electronic version)
ISSN: 1404-1219

Department of Computer Science
University of Antwerp
Middelheimlaan 1
2020 Antwerp
Belgium

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Cover art: "June's multi-colored eyes" by Keith Kissel, CC BY 2.0
Typeset using LaTeX
Printed in Sweden by Tryckeriet i E-huset, Lund, 2023

Mutation Testing: Fewer, Faster, and Smarter

Sten Vercammen

Supervisors:

Prof. Dr. Serge Demeyer Prof. Dr. Görel Hedin
Dr. Markus Borg

Thesis submitted for the degree of:
- Doctor of Science: Computer Science[1]
- Doctor of Philosophy in Engineering[2]

[1]University of Antwerp, Belgium
[2]Lund University, Sweden

Acknowledgements

The journey of completing a PhD is not an easy one, it requires a lot of hard work, determination and support from loved ones, mentors and colleagues. My own journey was no exception, and I am incredibly grateful for the support and guidance I received from my supervisors Serge Demeyer, Markus Borg, and Görel Hedin, as well as my colleagues, my family, and my friends throughout the years.

Serge Demeyer has been an incredible mentor, even before the start of my PhD. He taught us the fundamentals of software engineering, computer science, and how to perform research. In this regard, I would also like to thank Ali Parsai who introduced me to mutation testing and awakened my passion for that topic. Their guidance, knowledge and experience in the field have been fundamental in shaping my research and providing me with the necessary tools to succeed.

Markus Borg has been a vital mentor and support system throughout my PhD. His expertise and experience in empirical research have helped me navigate the complex landscape of my research studies and his insights have been elementary for my own research.

Görel Hedin has been an invaluable asset to my work. I am deeply grateful to her for imparting her knowledge and expertise on compilers to me, which has been instrumental in my understanding, growth, and contribution to the mutation testing field.

I am incredibly grateful for the time and energy my supervisors invested in me, and for the trust they placed in my abilities. Thanks to them I have had the privilege to work with excellent industrial partners to validate my research. I would like to thank them as well, as without them, my research would only be theoretical.

I would also like to thank my family and friends for their support, encouragement, and belief in me, throughout this journey. I am also grateful for their understanding whenever my studies and PhD distracted or prevented my attendance at social events.

I would like to express my sincerest gratitude to my girlfriend Sarah Pinseel, for her unwavering love, support, and encouragement throughout my PhD journey. I am utterly grateful for her understanding and patience during the long nights and weekends spent working on my research. Especially as we decided to completely renovate our house during my PhD, which took away almost all remaining leisure time. I am truly blessed to have her by my side and I am deeply grateful for her love and support.

Finally, I would like to express my gratitude to the FWO: Research Foundation – Flanders who funded my research, as without them this PhD would not have been possible.

EN: Abstract

The growing reliance on automated software tests raises a fundamental question: How trustworthy are these automated tests? Today, mutation testing is acknowledged within academic circles as the most promising technique for assessing the fault-detection capability of a test suite. The technique deliberately injects faults (called mutants) into the production code and counts how many of them are caught by the test suite.

Mutation testing shines in systems with high statement coverage because uncaught mutants reveal weaknesses in code which is supposedly covered by tests. Safety-critical systems –where safety standards dictate high statement coverage– are therefore a prime candidate for mutation testing. In safety-critical software, C and C++ dominate the technology stack. Yet this is not represented in the mutation testing community: a systematic literature review on mutation testing from 2019 analysed 502 papers and reported that from the 190 empirical studies, 62 targeted the C language family and out of the 76 mutation testing tools, only 15 targeted the C language family. Despite the apparent potential, mutation testing is difficult to adopt in industrial settings, because the technique —in its basic form— requires a tremendous amount of computing power. Without optimisations, the entire code base must be compiled and tested separately for each injected mutant. Hence for medium to large test suites, mutation testing without optimisations becomes prohibitively expensive.

To make mutation testing effective in an industrial setting, we set three objectives: (1) generate fewer mutants, (2) process them smarter and (3) execute them faster. To meet our objectives, we investigate the most promising techniques from the current state-of-the-art. This ranges from leveraging cloud technology to compiler integrated techniques using the Clang front-end. These optimisation strategies allow to eliminate the compilation and execution overhead in order to to support efficient mutation testing for the C language family.

As a final step, we perform an empirical study on the perception of mutation testing in industry. The aim is to investigate whether the advances are sufficient to allow industrial adoption and to identify any remaining barriers preventing industrial adoption.

In this Ph.D. thesis we show that a combination of mutation testing optimisation techniques from the *do fewer*, *do faster*, and *do smarter* are needed to perform mutation testing in a continuous integration setting. Furthermore, the industrial perception of mutation testing is evolving as additional organisations recognise its potential.

NL: Samenvatting

De groeiende afhankelijkheid van geautomatiseerde softwaretesten roept een fundamentele vraag op: hoe betrouwbaar zijn deze geautomatiseerde testen? In academische kringen wordt *mutation testing* de dag van vandaag erkend als de meest veelbelovende techniek om het vermogen van een testbatterij om fouten te detecteren, te beoordelen. De techniek injecteert opzettelijk fouten (genaamd mutanten) in de productiecode en telt hoeveel ervan door de testbatterij worden gevonden.

Mutation testing blinkt uit in systemen met een hoge instructiedekking, omdat niet-gevangen mutanten de zwakke plekken in code onthullen die zogezegd door de testen worden gedekt. Kritische systemen – waar de veiligheidsnormen een hoge instructiedekking voorschrijven – zijn daarom een uitstekende kandidaat voor *mutation testing*. In veiligheidskritieke softwaresystemen domineren C en C++ de technologiestack. Echter is dit niet vertegenwoordigd in het veld van *mutation testing*: een systematisch literatuuroverzicht over *mutation testing* uit 2019 dat 502 papers analyseerde, meldt dat van de 190 empirische studies, 62 zich richten op de C-taalfamilie en van de 76 *mutation testing* programma's er maar 15 zich richten op de C-taalfamilie. Ondanks het schijnbare potentieel, is *mutation testing* moeilijk toe te passen in een industriële omgevingen. Dit omdat de techniek — in zijn basisvorm — een enorme hoeveelheid rekenkracht vereist. Zonder optimalisaties moet voor elke geïnjecteerde mutant het volledige project afzonderlijk worden gecompileerd en getest. Hierdoor wordt *mutation testing* zonder optimalisaties voor middelgrote tot grote projecten onpraktisch.

Om *mutation testing* mogelijk te maken in een industriële omgeving, hebben we drie doelen gesteld: (1) minder mutanten genereren, (2) ze slimmer verwerken en (3) ze sneller uitvoeren. Hiervoor zullen we de meest veelbelovende van de huidige grensverleggende technieken onderzoeken. Dit varieert van het gebruik van cloudtechnologie tot compiler geïntegreerde technieken met behulp van de *Clang front-end*. Deze optimalisatiestrategieën maken het mogelijk om de compilatie- en uitvoeringsoverhead te elimineren en zo efficiënte *mutation testing* voor de C-taalfamilie te ondersteunen.

Als laatste stap voeren we een empirisch onderzoek uit naar de perceptie van mutatietesten in de industrie. Het doel is om te onderzoeken of de vooruitgang voldoende is om industriële adoptie toe te staan en om de eventuele potentiële resterende barrières te identificeren die de industriële adoptie voorkomen.

In deze Ph.D. thesis tonen we dat een combinatie van *mutation testing* optimalisatietechnieken die minder mutanten genereren, ze slimmer verwerken en ze sneller uitvoeren nodig is om *mutation testing* uit te voeren in een *continuous integration* omgeving. Verder, evolueert de industriële perceptie van *mutation testing* naarmate meer organisaties het potentieel ervan erkennen.

SE: Sammanfattning

Ökat användande av testautomation föranleder en grundläggande fråga: Hur tillförlitliga är egentligen alla dessa automatiserade tester? Inom den akademiska forskningen anses mutationstestning vara den mest lovande tekniken för att bedöma en testsvits förmågas att upptäcka fel. Tekniken introducerar avsiktligt fel (så kallade mutanter) i produktionskoden och utvärderar hur många av felen som upptäcks av testsviten.

Mutationstestning är särskilt användbart för källkod med hög grad av kodtäckning. Detta beror på att mutanter som inte upptäcks avslöjar testfalls bristande förmåga att upptäcka fel. Saknas kodtäckning finns inte heller någon anledning att utvärdera testfallen. Säkerhetskritiska system - för vilka säkerhetsstandarder kräver hög grad av kodtäckning - är därför lämpliga kandidater för mutationstestning. I säkerhetskritisk mjukvara dominerar C och C++ teknikstacken. Detta återspeglas inte i mutationstestningsforskningen. En systematisk översiktsstudie från 2019, baserad på 502 artiklar, rapporter att bara 62 av 190 empiriska studier betraktar C-språksfamiljen. Vidare rapporterades att enbart 15 av 76 identifierade mutationstestningsverktyg behandlar källkod från C-språksfamiljen. Trots den uppenbara potentialen har mutationstestning visat sig svårt att införa i industriella utvecklingsmiljöer, eftersom tekniken - i sin grundform - kräver en enorm mängd beräkningskraft. Utan optimeringar måste hela kodbasen kompileras och testas separat för varje introducerad mutant. Av denna anledning blir mutationstestning utan optimeringar i praktiken oanvändbart för medelstora till stora testssviter.

För att göra mutationstestning effektivt i en industriell utvecklingsmiljö sätter vi tre mål: (1) generera färre mutanter, (2) bearbeta dem smartare och (3) exekvera dem snabbare. Vi undersöker de mest lovande teknikerna från forskningsfronten. Till exempel, molnteknologi och kompilatorbaserade tekniker som använder Clang-front-enden. Dessa optimeringsstrategier leder till eliminering av kompilerings- och exekveringsoverhead vilket möjliggör resurseffektiv mutationstestning för C-språksfamiljen.

Avslutningsvis genomför vi en empirisk studie av industriella perspektiv på mutationstestning. Syftet är att utvärdera om optimeringarna är tillräckliga för industriella kontexter samt att identifiera eventuellt återstående hinder för storskalig tillämpning. Våra resultat visar att industrins syn på mutationstestning har utvecklats efter hand som fler utvecklingsorganisationer upptäckt möjligheterna med tekniken.

Avhandlingen demonstrerar att en kombination av optimeringarna är nödvändiga för att tillämpa mutationstestning i industriella *continuous integration*-kontexter. Avslutningsvis visar vi att industrins syn på mutationstestning utvecklas positivt efter hand som fler organisationer värdesätter dess potential.

Popular Summary

Speeding up Mutation Testing for a Superior Alternative to Code Coverage

Sten Vercammen, *Dept. of Computer Science, Lund University*
Dept. of Computer Science, University of Antwerp

Nowadays, many programmers tend to write automated tests for their projects. They then decide whether their project is sufficiently tested by running a code coverage technique. This tells us which parts of the project were executed during testing. But, this is exactly and only what these code coverage techniques can tell you. It cannot tell you that your tests are actually testing the code. For all we know, the executed code might not even be verified in the test cases.

Even with 100% code coverage, all you know is that the statements in your project have been executed without issues. It does not guarantee the absence of faults.

> Mutation testing measures the *fault detection capacity*

This is where mutation testing comes into play. Today, it is the state-of-the-art technique for assessing the *fault-detection capacity* of a test suite by deliberately injecting defects (called *mutants*) into the production code and counting how many of them are caught by the test suite. The more mutants the test suite can detect, the higher its fault-detection capability is. Where mutants are uncaught, additional tests need to be written.

Mutation testing shines in systems with high statement coverage because uncaught mutants reveal weaknesses in code which are supposedly covered by tests. Safety-critical systems –where safety standards dictate high statement coverage– are therefore a prime candidate for validating optimisation strategies. In safety-critical software, C and C++ dominate the technology stack. Yet in the mutation testing community, the C language family is somehow neglected: a systematic literature review on mutation testing from 2019 reports that less than 25% of the primary studies target source code from the C language family. This opens up opportunities as the C language family is a mature technology with considerable tool support available.

> Optimisations enable drastic speedups

Unfortunately, mutation testing is seldom adopted in practice, even less in an industrial setting, as it requires a tremendous amount of computing power. Without optimisations, the entire code base must be compiled and tested separately for each injected mutant. To combat this, we implemented a series of optimisation techniques that drastically reduce the execution time of mutation testing by generating *fewer* mutants, executing them *faster* and processing them *smarter*.

First, we exclude *invalid* mutants, as these mutants only cause compilation problems, there is no benefit in executing them. For this, we build our mutant generation tool in tandem with the semantic analyser of the Clang compiler, this allows us to verify which of our mutants are compile-time correct.

The two most time-consuming parts of a mutation analysis are 1: the compilation of the mutants and 2: the execution of the mutants.

> Mutant Schemata reduces compilation from days to minutes!

In order to speed up the first part, we implemented a *mutant schemata* technique which allows us to compile all mutants simultaneously, instead of compiling each mutant separately. Here, all mutants are inserted into the project code at once and additional functions are added to allow the activation of a single mutant at runtime. Excluding the invalid mutants is a necessity as if even a single mutant causes a compilation error, the complete mutation analysis will fail. This drastically reduces the compilation time and allows us to execute the mutants faster. We saw a reduction in compilation time **from 4 days to just 7 minutes**!

> A mutant is only reached by 10% of the test cases

This leaves us with the execution phase. Instead of executing the entire test suite for each mutant, we figured that we only need to execute the test cases that actually reach the mutant. We then created a tool that tells us just that, by instrumenting the code base to emit which mutant is reached and executing each test case separately. It turns out that, on average, only 10% of the test cases actually reach the mutant. This reduction allows us to execute the mutants up to 10 times faster.

We can process the mutants smarter by exploiting state-space information. Instead of letting each mutant initiate execution from the start of the program, each mutant is started from the mutation point itself by forking the process, essentially avoiding redundant executions. This can achieve an additional speedup of 2 to 3 times.

Finally, we implemented a cloud solution to distribute the mutation workload over a cloud infrastructure. Here we saw that by doubling the number of hardware nodes, the execution time almost halves, nearly increasing the speed-up linearly.

> Industry wants to know their test effectiveness

Thanks to these speedup techniques, the mutation analysis has been sped up drastically. In order to evaluate whether the speedups were sufficient for an industrial environment, we ran a small-scale empirical study with different companies. We set up a mutation analysis and asked their opinions about mutation testing. All companies were very interested about the results as they wanted to know their test effectiveness. The companies are positive about the capabilities of mutation testing and all acted upon the analysis results. The empirical study led to an increased test effort within all participating companies. Some of them continue to run the mutation analysis within some departments and evaluate whether they can extend the technique to others.

We can thus safely say that the mutation testing technique is starting to have an impact in the industry and that more and more companies are looking into mutation testing as a superior alternative to code coverage.

List of Publications

Papers included in the Ph.D. thesis

A Sten Vercammen, Serge Demeyer, Markus Borg, and Sigrid Eldh. Speeding up mutation testing via the cloud: lessons learned for further optimisations. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–9, Oulu Finland, October 2018. ACM. ISBN 9781450358231. doi: 10.1145/3239235.3240506. URL `https://dl.acm.org/doi/10.1145/3239235.3240506`

B Sten Vercammen, Mohammad Ghafari, Serge Demeyer, and Markus Borg. Goal-oriented mutation testing with focal methods. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 23–30, Lake Buena Vista FL USA, November 2018. ACM. ISBN 9781450360531. doi: 10.1145/3278186.3278190. URL `https://dl.acm.org/doi/10.1145/3278186.3278190`

C Sten Vercammen, Serge Demeyer, and Lars Van Roy. Focal methods for C/C++ via LLVM: steps towards faster mutation testing. In *Proceedings of the 20th Belgium-Netherlands Software Evolution Workshop, Virtual Event / 's-Hertogenbosch, The Netherlands, December 7-8, 2021*, volume 3071 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021

D Sten Vercammen, Serge Demeyer, Markus Borg, Niklas Pettersson, and Görel Hedin. Mutation Testing Optimisations using the Clang Front-end. 2022. doi: 10.48550/ARXIV.2210.17215. URL `https://arxiv.org/abs/2210.17215`
- under revision in *Software Testing, Verification and Reliability 2023*

E Sten Vercammen, Serge Demeyer, Markus Borg, and Pettersson. F-ASTMut Mutation Optimisations Techniques using the Clang Front-end. 2023
- under revision in *Software Impacts 2023*

F Sten Vercammen, Serge Demeyer, Markus Borg, and Pettersson. Mutation Testing Requirements Elicitation in Industry. 2023
- Submitted for peer review

Papers not included in the Ph.D. thesis

1. Serge Demeyer, Ali Parsai, Sten Vercammen, Brent van Bladel, and Mehrdad Abdi. Formal Verification of Developer Tests: A Research Agenda Inspired by Mutation Testing. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles*, volume 12477, pages 9–24. Springer International Publishing, Cham, 2020. ISBN 9783030614690 9783030614706. doi: 10.1007/978-3-030-61470-6_2. URL `http://link.springer.com/10.1007/978-3-030-61470-6_2`

2. Zhong Xi Lu, Sten Vercammen, and Serge Demeyer. Semi-automatic Test Case Expansion for Mutation Testing. In *2020 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, pages 1–7, London, ON, Canada, February 2020. IEEE. ISBN 9781728162713. doi: 10.1109/VST50071.2020.9051637. URL `https://ieeexplore.ieee.org/document/9051637/`

3. Sten Vercammen, Serge Demeyer, Markus Borg, and Robbe Claessens. Flaky Mutants; Another Concern for Mutation Testing. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 284–285, Porto de Galinhas, Brazil, April 2021. IEEE. ISBN 9781665444569. doi: 10.1109/ICSTW52544.2021.00054. URL `https://ieeexplore.ieee.org/document/9440140/`

Contribution Statement

All papers included in the thesis have been coauthored with other researchers. Below we list the contributions of each individual author.

A Four authors contributed to this paper. The first author, Sten Vercammen, conceptualised the paper, developed the software, curated the experiment data, performed the analysis of the data, and wrote the paper. The remaining authors reviewed and edited the paper, whilst Serge Demeyer also supervised the project.

B Four authors contributed to this paper. In this paper the ideology of an existing, non-mutation related tool from Mohammad Ghafari, the second author, was investigated as a potential technique to speed up mutation testing. Mohammad Ghafari provided the details about the underlying technique. The first author, Sten Vercammen, conceptualised the paper, curated the experiment data, performed the analysis of the data, and wrote the paper. Serge Demeyer and Markus Borg reviewed and edited the paper, whilst Serge Demeyer also supervised the project.

C Four authors contributed to this paper. This paper includes the preliminary research of a master student thesis conducted by Robbe Claessens, who is the fourth author. The research was conceptualised, reviewed, and supervised by the first author Sten Vercammen. Additional research and evaluation was performed to complete the paper. The paper is conceptualised and written by the first author Sten Vercammen. Serge Demeyer and Markus Borg reviewed and edited the paper, whilst Serge Demeyer also supervised the project.

D Five authors contributed to this paper. Here, Sten Vercammen conceptualised the paper, implemented the proof-of-concept tool, curated the experiment data, performed the analysis of the data, and wrote the paper. Niklas Pettersson ran the experiments on the SAAB project and aided with the experimental integration into Dextool mutate. The remaining authors reviewed and edited the paper, whilst Serge Demeyer and Markus Borg also supervised the project.

E Three authors contributed to this paper. This paper describes the research tool developed and used to analyse the mutation testing optimisations. The tool itself was conceptualised and developed by the first author Sten Vercammen. The paper was also written by the first author. The second and third authors, Serge Demeyer and Markus Borg, reviewed the paper.

F Three authors contributed to this paper. This paper describes the empirical study about the industrial perspective on mutation testing and whether the recent advances regarding the speedup are sufficient for industrial consideration. The paper was conceptualised and written by the first author Sten Vercammen. The second author, Markus Borg, aided in the review and analysis of the data. The third author Serge Demeyer reviewed and edited the paper.

Contents

List of Figures

List of Tables

List of Algorithms

List of Code Listings

Chapter 1

Introduction

Software testing is the dominant method for quality assurance and quality control in software development organisations [10, 11]. Software testing was established as a disciplined approach in the late 70's when it was defined as *"executing a program with the intent of finding an error"* [12]. In the last decade, this intent shifted dramatically with the advent of continuous integration [13]. Many software tests are now fully automated, and serve as quality gates, safeguarding against programming faults. The scale at which automated software tests are adopted in modern software organisations is mind-boggling. Microsoft for instance reported that approximately 11 months of development on Windows comprised more than 30 million test executions. Google, on the other hand, reported that *"In an average day, TAP integrates and tests [. . .] more than 13K code projects, requiring 800K builds and 150 Million test runs."* [14]. As a result of this continuous integration approach, software organisations are capable of releasing faster. Tesla, for example, uploads new software in its cars once every month [15]. Amazon pushes new updates to production every 11.6 seconds [16].

1.1 Motivation

The growing reliance on automated software tests raises a fundamental question: How trustworthy are these automated tests? For effective testing, software teams need strong tests which maximise the likelihood of exposing faults [12]. Traditionally, the strength of a test suite is assessed using code coverage, revealing which statements are poorly tested. However, code coverage is a poor indicator of test effectiveness as it cannot determine the *fault-detection capacity* of a test suite [17–19]. Moreover, stronger coverage criteria, like full MC/DC coverage (Modified Condition/Decision Coverage, a coverage criterion often mandated by functional safety standards that target critical software systems, e.g., ISO 26262 and DO-178C) still do not guarantee the absence of faults [20, 21]. Hence, alternatives are being investigated. Today, mutation testing is acknowledged within academic circles as the most promising technique for assessing the *fault-detection capability* of a test suite [22, 23]. The technique deliberately injects faults (called mutants) into the production code and counts how many of them are caught by the test suite. The more mutants the test suite can detect, the higher its *fault-detection capability* is – referred to as the *mutation coverage* or *mutation score.* The mutants generated by mutation testing closely resemble actual faults [24, 25]. Furthermore, mutation testing has been shown to be superior to simple code coverage metrics in discovering test suite weaknesses [26].

Alternative techniques like fault injection, fuzzing, and model-driven testing are valid testing approaches with different purposes. Each of them are used to test and improve different aspects of the code. They are complementary to mutation testing. Where mutation testing measures the fault-detection capabilities of the test suite, fault injection investigates the system's reaction to faulty behaviour [27, 28]. Fuzzing examines the system's behaviour when invalid or unexpected inputs are provided [29, 30]. Model-driven testing compares the current behaviour of the system to the expected behaviour of the model [31, 32].

Case studies with safety-critical systems demonstrate that mutation testing could be effective where traditional structural coverage analysis and code inspections have failed [33, 34]. Google, on the other hand, reports that mutation testing provides insight into poorly tested parts of the system, but –more importantly– also reveals design problems with components that are difficult to test, hence must be refactored [35]. In a similar vein, a blog post from a software engineer at NFluent, comments on integrating Stryker (a mutation tool for .Net programs) in their development pipeline [36]. There as well, mutation testing revealed weaknesses in the test suite but also illustrated that refactoring allowed for simpler test cases which subsequently increased the mutation score.

Mutation testing shines in systems with high statement coverage because of uncaught mutants reveal weaknesses in code which are supposedly covered by tests. Safety-critical systems –where safety standards dictate high statement coverage– are therefore a prime candidate for validating optimisation strategies. In safety-critical software, C and C++ dominate the technology stack [37]. Yet this is not represented in the mutation testing community: a systematic literature review on mutation testing from 2019 analysed 502 papers and reported that from the 190 empirical studies, 62 targeted the C language family and out of the 76 mutation testing tools, only 15 targeted the C language family [23]. In this Ph.D. thesis we use the term C language family to describe the C, Objective C, C++, Objective C++ and their variants. This opens up opportunities as the C language family is a mature technology with considerable tool support available.

Despite the apparent potential, mutation testing is difficult to adopt in industrial settings. One of the reasons is that the technique —in its basic form— requires a tremendous amount of computing power. Without optimisations, the entire code base must be compiled and tested separately for each injected mutant [22]. During one of our experiments with an industrial code base, we witnessed 48 hours of mutation testing time on a test suite comprising 272 unit tests and 5,256 lines of test code for a system under test comprising 48,873 lines of production code [1]. Hence for medium to large test suites, mutation testing without optimisations becomes prohibitively expensive. Therefore, the goal of this Ph.D. thesis is to investigate a series of optimisation techniques, which combined, drastically speed up mutation testing to enable effective mutation testing in an industrial setting. We, therefore, tailor the optimisation techniques to the C language family or to be language-independent.

Thesis Statement — Although mutation testing has been shown to be the most effective method for assessing the *fault-detection capability* of a test suite, its widespread adoption in industry has been limited due to the time-consuming nature of traditional, unoptimised approaches. By addressing this performance issue, mutation testing has the potential to become a widely-used technique for improving software quality.

1.2 Mutation Testing in a Nutshell

Mutation hypotheses. Mutation testing (sometimes also named *mutation analysis*) is based upon the following two fundamental hypotheses. The Competent Programmer Hypothesis states that programmers are competent, implying that they program close to the correct version, and that any faults that occur are due to small syntactical errors [38, 39]. The Coupling Effect states that test suites that can detect simple errors are also able to detect complex errors [39].

The RIPR model. The RIPR model (Reachability, Infection, Propagation, Reveal) states that in order to reveal a fault, a test case must a) reach the faulty statement (Reachability), b) cause the program state to become faulty (Infection), c) propagate the fault to the program output (Propagation) and d) cause a failure, i.e. the faulty state is asserted by the test case to its intended state (Reveal).

Weak, firm, and strong mutation testing. Three different kinds of mutation testing are linked to the RIPR model: *weak*, *firm*, and *strong*. For *weak* mutation testing, only the first two conditions of the RIPR model need to be satisfied. This means that a mutant is considered detected from the moment the program state of the original program and the mutated program differ. With *firm* mutation testing, an extension of *weak* mutation testing, the user can decide which component of the program state should differ from the original for a mutant to be considered as detected. Lastly, for *strong* mutation testing, all conditions of the RIPR model need to be satisfied. This means that a mutant must influence the observable output of the program (the test oracle). In this Ph.D. thesis we utilise *strong* mutation testing as empirical evidence has shown that it is more powerful than *weak* and *firm* mutation testing [40].

Killed / survived mutants. Mutation testing deliberately injects faults (called mutants) into the production code and counts how many of them are caught by the test suite. A mutant caught by the test suite, i.e. at least one test case fails on the mutant, is said to be *killed*. When all tests pass, the mutant is said to be *survived*.

Mutation Operators. Mutation testing mutates the program under test by artificially injecting a fault based on a mutation operator. A mutation operator is a source code transformation which introduces a change into the program under test. Typical examples are replacing a conditional operator (e.g., $>=$ into $<$) or an arithmetic operator (e.g., $+$ into $-$).

Invalid Mutants. Mutation operators introduce syntactic changes, and hence may cause compilation errors in the process. If we apply the arithmetic mutation operator (AOR) to e.g. "a * b", then we get four mutants as shown in Listing 1.1. However, the modulo operator (%) will give an "`invalid operands to binary expression`" error, as the modulo operator is not defined for floating point data types. The mutant can thus not be compiled and is considered invalid.

Listing 1.1: Mutation Example

```
float f(float a, float b) {
    return a * b;    // original code
}
    return a + b;    // mutant 1
    return a - b;    // mutant 2
    return a / b;    // mutant 3
    return a % b;    // mutant 4
          ~~~^~~~Invalid operands to binary expression
```

Timed-out Mutants. Mutants can change the code in such a way that they can get stuck in an infinite loop, or increase their execution time drastically. Such changes would be caught by the continuous integration approach. To prevent such mutants from wasting too much time, these mutants are generally stopped, and labelled as killed, after a certain period of time, often a multitude of the original execution time.

Equivalent Mutants. Injected mutants can be syntactically different from the original software system, but semantically identical. These mutants do not modify the meaning of the original program, and can therefore not be detected by the test suite. They show up as survived mutants, but they need to be manually identified and labelled as equivalent mutants as they can never be killed. They waste developer time, as they need to be manually identified and labelled as equivalent mutants, because they show up as survived mutants. As they waste developer time, a big challenge of mutation is handling (and/or eliminating) these equivalent mutants. An overview of techniques to overcome the equivalent mutant problem has been provided by Madeyski et al [41].

Mutation Testing Process. To explain the time-consuming nature of the mutation testing process, Figure 1.1 shows the essential steps of a mutation analysis without any optimisations. The software system needs to build without errors and all software tests should succeed before mutation testing can even begin; this is called the *pre*-phase. Then, the two main phases are executed: *(A)* the generate mutants phase and *(B)* the execute mutants phase. In phase A, mutants are generated for all source files. In phase B, for each mutant, all tests are executed and the result —whether or not it was killed— is saved. Finally, all the results are gathered and the final report is created in the *post*-phase.

1.3 Related Work

A lot of research is devoted to optimising the mutation testing process, summarised under the principle — *do fewer*, *do smarter*, and *do faster* [42].

- *Do fewer* approaches minimise the execution time by reducing the total number of mutants to execute. Such an optimisation can be implemented by generating fewer mutants in phase A in Figure 1.1 or by selecting a subset of all mutants. An incremental approach [43], limiting the mutation analysis to code changed in a commit, is a particularly relevant example of a “do fewer” approach. A reduced set of mutants normally incurs an information loss compared to the full set of mutants, however, the effectiveness is often acceptable [22]. Nevertheless, there exist mutants

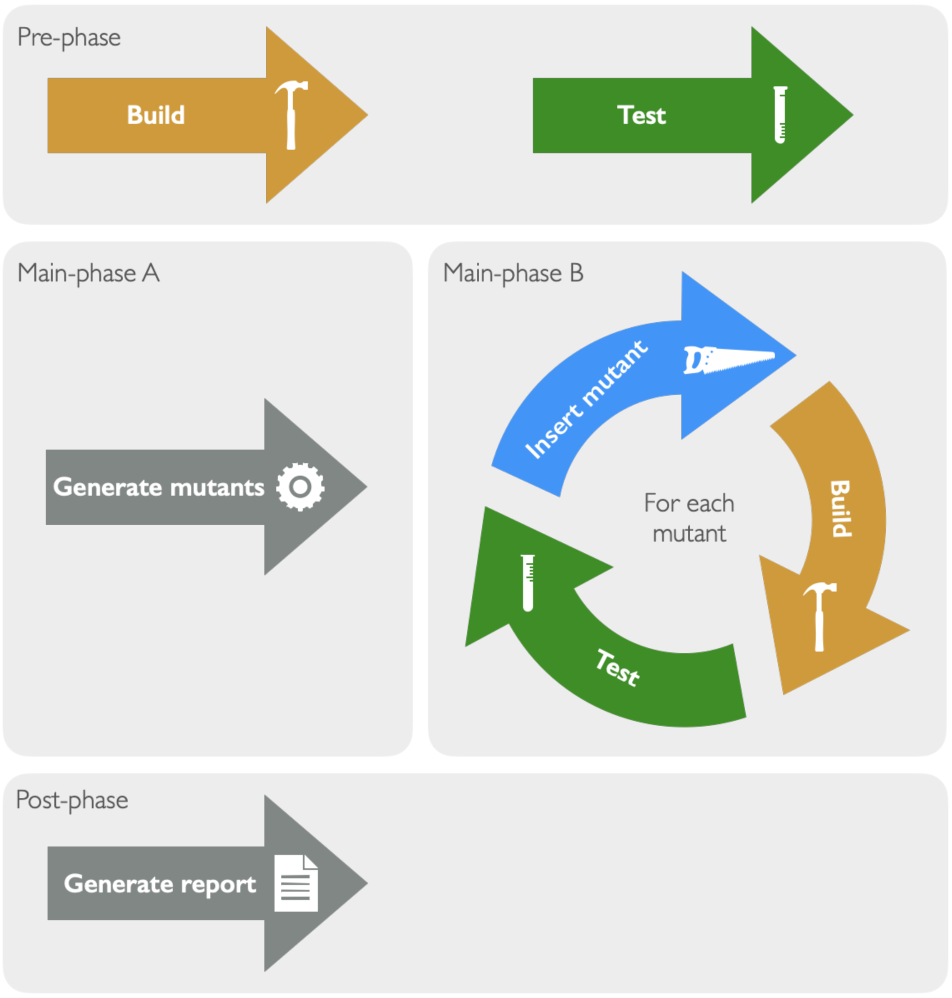


Figure 1.1: Phases of Unoptimised Mutation Testing

for which we can know the results before we execute them. Excluding them effectively reduces the number of mutants we need to execute without any information loss. These mutants include the *invalid mutants* which cause compilation errors and the so-called *unreachable mutants* which are mutants in code that the test suite does not cover, therefor they can never be detected.

- *Do faster* approaches attempt to minimise the execution time by reducing the execution cost for each mutant (in phase B in Figure 1.1). By design, a mutated program is almost identical to the original program which can be exploited during the compilation step. *Mutant schemata* [44] is the best known example. With this technique, all mutants get injected simultaneously (guarded by a global switch variable), hence the project is compiled only once (*Build* in phase B). During the mutant execution phase (*Test* in phase B) the global switch is used to select the actual mutant to execute. The execution time of a mutant can also be reduced with *test prioritisation* techniques. By rearranging the test suite, the tests with the highest likelihood of failure will be executed first, reducing the test suite run time using early-failure [45].

- *Do smarter* approaches attempt to minimise the execution time by exploiting the computer hardware (e.g. distributed architectures, vector processors, fast memory access). Each mutant (in phase B in Figure 1.1) has few data dependencies, hence can be executed in parallel. Parallel execution of mutants, either on dedicated hardware [46] or in the cloud [1] is known to speed up the process by orders of magnitude. *Split-stream mutation testing* is one example of a "do smarter" optimisation [47]. By retaining state information between test runs, split-stream mutation testing avoids the redundant execution of statements up until the mutation point.

The current state-of-the-art demonstrates the feasibility of mutant optimisations for the C language family under the principle *do fewer*, *do smarter*, and *do faster*. These are, however, investigated in isolation. There is no research that analyses these techniques together for the C language family. Furthermore, detailed measurements to investigate the impact of these techniques for each of the mutation steps are lacking.

Different approaches are often synergistic, where a combination of techniques becomes more than the sum of the parts. Some approaches are orthogonal to one another and are easy to combine. Excluding unreachable mutants, for example, can be combined with any other optimisation. Other approaches, however, may depend on each other. Mutant schemata, for instance, requires that all invalid mutants are excluded because even a single invalid mutant will immediately invalidate the whole mutated program. Measuring the speedup of a given optimisation strategy should take these synergies into account.

1.3.1 Distribution

Mutation testing has shown to be able to run in parallel on a distributed architecture [46, 48, 49]. Consequently, researchers are currently investigating cloud solutions to share the computational load across a series of hardware nodes. Hadoopmutator [50] and Eminent [51] are the ones we have found in the literature.

Hadoopmutator is a cloud-based Mutation Testing framework that is implemented using Hadoop's MapReduce[1]. During the mapping phase, each mutation operator is assigned to a separate node, creating a single mutant and executing the test suite. The subsequent reduce phase aggregates the results from all the test executions and calculates the mutation score.

Data transfer between the nodes is handled using Hadoop's MapReduce and the Hadoop Distributed File System (HDFS™). Hadoopmutator is applied on two open source projects, and the authors report a speed-up of 9.57x and 12.83x using 13 nodes. For small projects like Apache Wicket, the authors state that "*the overhead of running Hadoop on the compute nodes becomes significant relative to the time needed to generate and executes the tests and hence the optimal performance gain is not attained*". The influence of speed-up with large projects that affect the I/O and network traffic was not investigated.

[1] `https://hadoop.apache.org`

Eminent (EMbarrassINgly parallEl mutatioN Testing) is a distributed Mutation Testing tool that relies on the Message Passing Interface standard for portable message-passing in parallel computing architectures [51]. Eminent uses *test-level granularity*, the test suite is split up and each test/mutant combination is run separately. If a single test (of a mutant) fails, the mutant is killed and all other remaining tests (related to that mutant) are canceled.

Eminent handles data transfer between the master and the nodes by means of a shared database. The test cases are sent to the worker processes which will execute them against the mutants and send the results back to the master, which compares them to the original. Eminent is applied on three projects, and the authors report a speed-up between 8x and 22x using 32 nodes. Large projects can have an impact on the scalability "*because of the high volume of network and I/O traffic generated by this application, which acts as a system bottleneck in the database node*".

1.3.2 Test Suite Reduction

Researchers have sought mitigation strategies for the computationally expensive mutation analysis [22]. Many approaches originate in work on test suite minimisation, a set-cover problem that has been shown to be NP-complete – but several approximation solutions have been proposed [52]. For example, Jeffrey and Gupta presented a test suite reduction technique with selective redundancy, a slightly more conservative approach (i.e. less reduction) that retains more of the fault detection effectiveness of the original test suite compared to previous work. Nevertheless, test suite reduction always requires a trade-off between execution time and fault detection effectiveness [53].

Several regression test selection methods have been proposed to speed up mutation testing, aiming at restricting test case execution to those that target the code changes. Regression test selection methods are either dynamic (i.e. using execution information) or static (i.e. based entirely on source code analysis). Chen and Zhang performed an extensive empirical evaluation of several state-of-the-art regression test selection methods for mutation testing on 20 GitHub projects [54], and concluded that the techniques are generally feasible on a file level but not for finer-grained analysis. Also, the methods studied are intended for evolving systems and not for a single version of source code.

Zhang *et al.* focused on speedup of mutation testing that works for a single source code version [45]. They developed FaMT (Faster Mutation Testing) as an approach to prioritise and reduce the number of test cases to execute for each mutant. Inspired by research on regression test prioritisation, FaMT reorders the test cases in a way to kill the mutant earlier. Subsequently, inspired by previous work on test suite reduction, FaMT runs only the subset of test cases with a high likelihood to kill the mutant. Thus, FaMT might under-approximate the mutation score – some of the skipped test cases might indeed have killed the mutant if they were executed.

There are also other approaches to exclude test cases from a test suite targeting a specific mutant. Bardin *et al.* proposed program verification to exclude test cases that cannot reach the mutant and/or that cannot infect the program state [55]. Other authors have explored using (static) symbolic execution techniques to identify whether a test case can detect mutants [56, 57]. An example of a tool implementing this approach is PIT [58] that executes only those test cases that have a chance to kill the mutant, i.e. the test cases that execute the faulty statement (thus fulfilling Reachability).

1.3.3 Compiler Integration

To reduce the computational cost of mutation testing, researchers have looked at compiler integration techniques. These techniques prevent the need to compile each mutant individually [44]. The two main approaches to achieve this are by either manipulating the source code, allowing the selection of the mutant at run time, or by manipulating the compiled program, i.e. the bitcode by injecting the mutant before executing it.

Mull is an open-source mutation testing tool[2] which modifies fragments of the bitcode. It only needs to recompile the modified fragments in order to execute the mutants, keeping the compilation overhead low [59]. Mull includes a *do-fewer* optimisation where you can limit which mutants are executed to only those mutants that are within a certain call-depth starting from the test case.

AccMut also modifies the bitcode to the compilation overhead low [60]. On top of that, it reduces redundant execution statements anywhere in the program by analysing the original and mutated program. It identifies the redundant statements by inspecting the (local) state of both programs. When they are identical, all following statement executions are identical and redundant until the next different statement. They have demonstrated an average speedup of 8.95x over a mutant schemata approach [60].

WinMut is a further evolvement of AccMut. By grouping mutants, instead of analysing each mutant individually, WinMut is able to reduce the amount of processes it spawns and reduces the the high overhead introduced by the interpreter. Their evaluation achieves an average speedup of 5.57x on top of AccMut.

Dextool is an open-source framework created for testing and static analysis of (often safety-critical) code. The Dextool framework is used within industry, for example within Saab Aeronautics. One of the plugins in the framework is Dextool mutate. It was developed with a heavy emphasis on the reporting part of mutation testing in order to better understand the output of mutation testing and to gain more insight into the project under test.

Dextool mutate (textually) analyses the source code for points to mutate and stores them in a central database. A distributed setup can be used by utilising multiple nodes which can each execute a subset of all mutants stored in the database. During this Ph.D. thesis, we created a proof-of-concept schemata plugin for Dextool mutate. This allowed us to work with the creators of Dextool to analyse the effects of the schemata plugin. As a result, the creators of Dextool created a proper implementation for mutant schemata into the tool.[3,4]

[2]`https://github.com/mull-project/mull`

[3]`https://github.com/joakim-brannstrom/dextool/tree/master/plugin/mutate/contributors.md`

[4]`https://github.com/joakim-brannstrom/dextool/blob/master/plugin/mutate/doc/design/notes/schemata.md`

1.4 Research Objectives

To make mutation testing effective in an industrial setting, we set three objectives: (1) generate *fewer* mutants, (2) execute them *faster* and (3) process them *smarter*. For each of these objectives, we investigate the most promising techniques from the current state-of-the-art and implement them into a proof-of-concept tool. As a final step, we perform an empirical study on the perception of mutation testing in industry. The aim is to investigate whether the advances are sufficient to allow industrial adoption and/or identify any potential remaining barriers preventing industrial adoption.

Goal 1 – *generate fewer mutants by excluding mutants that do not add value*
In this Ph.D. thesis we implement an optimisation technique that during the generation phase excludes invalid mutants, i.e. mutants that would cause compilation errors. Additionally we implement a technique to exclude mutants that are not reached by the test suite. This reduces the number of mutants that needs to be considered for the mutation testing analysis.

Goal 2 – *execute mutants faster by reducing the execution and compilation overhead*
Not every mutant is covered by the entire test suite. A test case that does not reach the injected mutant does not add any value to the mutation analysis. We therefore implement both a statical and a dynamical technique to reduce the test set for each mutant, effectively reducing the execution overhead.

Instead of compiling each mutant individually, we implement a compiler integrated technique to compile all mutants at once. This drastically speeds up the compilation time. By instrumenting the source code of the project, mutants can be activated at run time.

Goal 3 – *process mutants smarter by exploiting the computer infrastructure*
We implement a scalable cloud-based technique to share the computational load of a mutation analysis across a series of hardware nodes. Each mutant is inherently independent from each other, making them an ideal candidate for distribution.

In a conventional tool implementation, each mutant always starts its execution from the same location, i.e. the start of the test case. It then executes the same path right until the actual mutant. This means that many redundant statements are being executed. To counter this, we implement a split-stream technique that exploits the state-space information to start the mutant from its mutation point instead of from the start of the test case.

Goal 4 – *validate mutation testing advances with the intent of industrial adoption*
To validate our optimisation techniques, we perform an empirical study in industry to understand the perceptions and attitudes of professionals in the field towards mutation testing. The aim is to investigate whether the mutation testing advances are sufficient to allow industrial adoption. By collecting empirical data on the level of awareness and understanding of the technique, as well as the factors that influence its perceived value and utility, we aim to provide insight into the adoption and implementation of mutation testing in industry. The results of this investigation will inform the development of strategies for

promoting the wider use of mutation testing, and identify areas for improvement and potential barriers to industrial adoption.

1.5 Research Approach

In this Ph.D. thesis we follow an iterative approach with a build-evaluate loop to speed up the mutation testing analysis. Our general work flow is depicted in Figure 1.2. This image is based on the design science research cycles [61]. We start from an (unoptimised) mutation testing tool. Based on the application domain, we then select the most promising optimisation techniques available from the current state-of-the-art and identify expected improvements and potential side-effects. In order to do so, we create a proof-of-concept implementation as part of a build-evaluate loop. We then validate the optimisations using a pilot study. This allows us to identify when unintended side effects occur and measure their impact. We then report these findings and publish the created artefact.

At this point, we start the cycle again by investigating which optimisation technique would best align with our mutation tool and speed it up further.

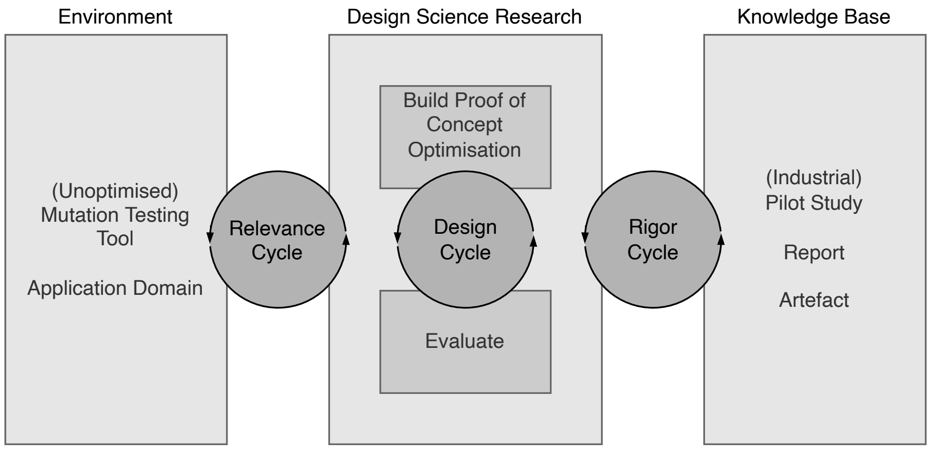


Figure 1.2: Research Approach
Image adapted from Hevner et. al [61]

1.6 Roadmap Optimisation Techniques

Figure 1.3 represents an overview of the developed techniques during this Ph.D. thesis and how they are linked to each other.

Paper A (p. 29) – We first implemented a cloud-based approach called DiMuTesTas to speed up the mutation testing analysis by distributing the workload over multiple computers. We find that by doubling the number of hardware nodes, the execution time almost halves, nearly increasing the speed-up linearly.

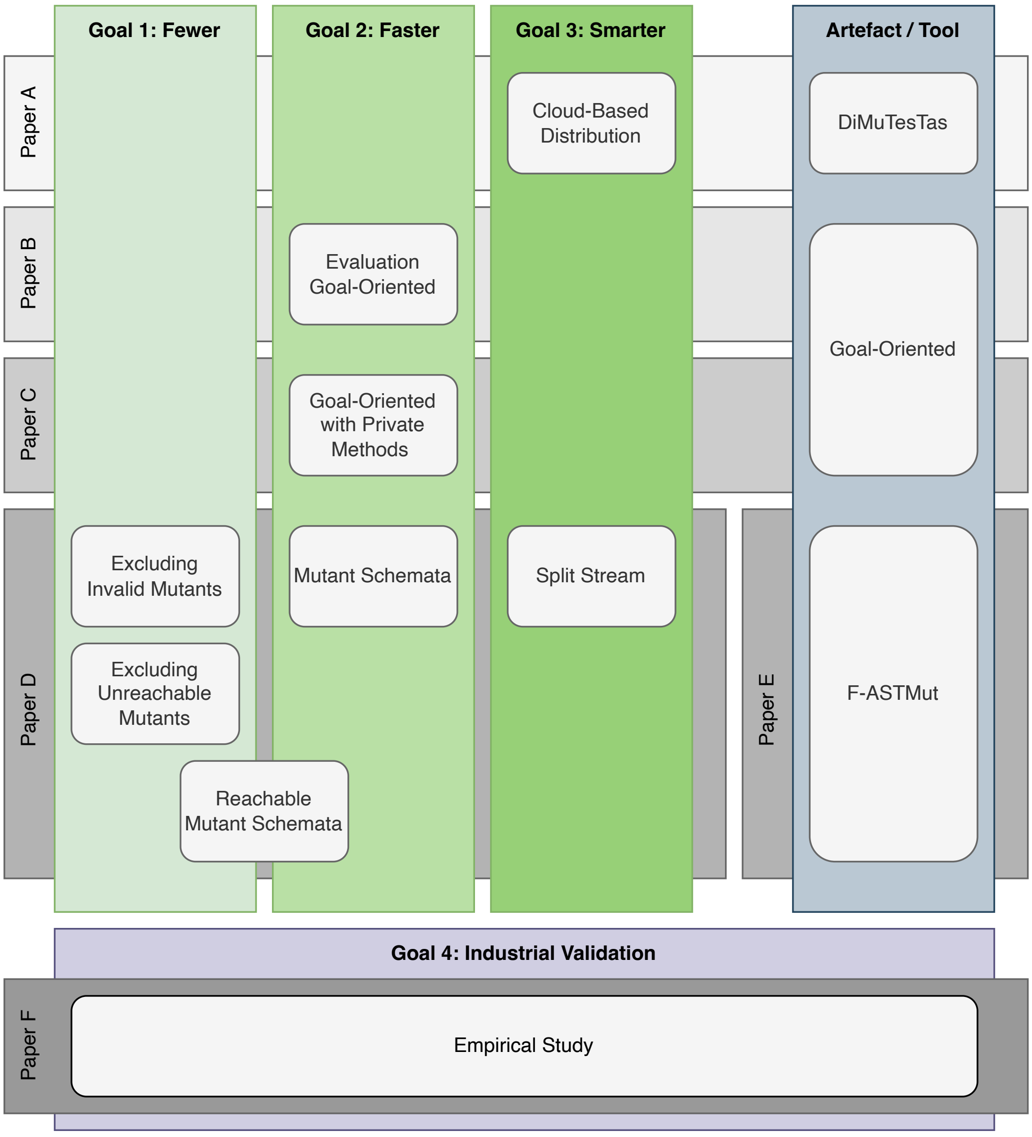


Figure 1.3: Roadmap Optimisation Techniques

Unoptimised mutation testing has two main time consuming phases: the first is the compilation time as each mutant is injected individually and compiled separately, the second is the execution time as all the test cases need to be ran against the mutant.

Paper B (p. 47) – To reduce the execution time of each mutant, we aim to reduce the test set which needs to be executed for each mutant to only those tests which are responsible for testing the methods in which the mutant resides. For this we validated the novel *goal-oriented mutation technique* and but found the lack of private methods support limiting its accuracy and potential speedup.

Paper C (p. 61) – We then created a proof-of-concept implementation of the *goal-oriented mutation technique* for the C language family including private method support. While we see promising execution speedups of 575x, additional work needs to be done before the technique can be used in practice, as most importantly, the static analysis does not yet support pointers. However, the technique is mature enough so that it can be used as a test prioritisation technique.

In order to reduce the test set for each mutant we look at an alternative approach using dynamic analysis. This allows us to trace which test cases are actually covering which mutants, allowing us to exclude the test cases that do not reach the mutant, effectively reducing the test set. The reduced test set gained by this technique is larger than with the goal-oriented approach as instead of covered test the goal-oriented approach aims to only select those test cases with the intent to test the methods in which the mutant resides. For the dynamic analysis we rely on the Clang front-end to instrument the code base. This also allows us to investigate and enable other optimisations like excluding invalid and unreachable mutants.

These optimisations reduce the execution time of each mutant. The compilation time then takes up the largest amount of the mutation analysis. To reduce this we again look at the Clang front-end to implement a compiler-integration technique called mutant schemata.

Paper D (p. 69) – The Clang front-end allows us to implement a variety of optimisations. The first optimisation reduces the total number of mutants that need to be executed by *excluding the invalid and unreachable mutants* mutants. Whilst generating the mutants, we can detect the *syntactically and semantically* invalid mutants, i.e. mutants that would cause compilation issues, by utilising the semantic analyser of the Clang front-end. Compared to an *unoptimised* approach, this speeds up the mutation analysis by a factor between 1.07x and 1.12x. While we can exclude the mutants that are not executed by any of the test cases, we go a step further. By instrumenting the code base, we can utilise a dynamic analysis to build a mutant to test relationship. This allows us to reduce the test set to only those test cases that execute the mutant. From our measurements, we have seen that the average number of mutants reached per test is between 10 and 20% of all valid mutants, implying a speedup between 5x and 10x.
The second optimisation presents a *mutant schemata* technique which virtually eliminates the compilation overhead by compiling all mutants simultaneously instead of compiling once for each mutant.
The third optimisation presents the *reachable schemata* technique. This is a combination of the above techniques, reducing both the execution time as the compilation time of the mutation analysis. Compared to an *unoptimised* approach we achieve a maximum speedup of 23.45x and 30.52x.
The final optimisation presents a *split-stream mutation technique* which aims to reduce the execution overhead by starting the execution of the mutants from their mutation point instead of from the start of the test suite. While this technique could yield a speedup of a factor 2, the technique is difficult to implement in real-world projects due to the requirement that all dependencies need to be revertible to specific states of the execu-

tion. This might even be impossible in case where data is stored and modified in external databases.

Paper E (p. 107) – We published F-ASTMut, our open-source mutation testing research tool for mutation optimisations utilising the Clang front-end as an artefact. The tool is designed for detailed measurements, analysis, and tuning of optimisation techniques. It includes the optimisations from Paper D.

During our work, we noted some peculiar behaviour from some mutants. These mutants did not behaved deterministically. Moreover, they appear intermittently, making them hard to detect. We call these *flaky mutants* and investigated their ramifications [9]. We go into more details in Section 1.9. These *flaky mutants* also appear in industry as we have seen then during our empirical study.

Paper F (p. 117) – We performed an empirical study examining the industrial perspective on mutation testing, whether it provides sufficient benefits for industrial adoption to motivate its costs, whether the recent mutation testing advances are sufficient to meet the computational performance requirements for industrial adoption, and whether significant pain points remain to be tackled for widespread industrial use. We have seen that mutation testing contributes to a requirements-based testing process, sometimes even revealing flawed or poorly phrased requirements. We have seen that with optimisations, mutation testing can be integrated into the continuous integration server. Mutation testing helps with a shift-left testing strategy, but due to time constraints it may not be practical to attempt to kill all mutants in large-scale applications. In this study, we have seen that equivalent mutants are less of an issue than considered in academic circles. Flaky mutants, however, are a real concern that needs to be addressed when considering mutation testing in the long run. Furthermore, our study indicates that mutation testing is not a replacement for human code review, it is a useful tool for offloading the identification and correction of low-hanging fruit.

1.7 Contributions of the Thesis

In this section, we present an overview of the contributions we made to achieve our research objectives. We first present the research advances we made to contribute to the field of knowledge and then present how we achieved our research objectives.

1.7.1 Research Advances

Mutation optimisation techniques for the C language family. C and C++ dominate the technology stack in safety-critical software [37]. Yet this is not represented in the mutation testing community: a systematic literature review on mutation testing from 2019 analysed 502 papers and reported that from the 190 empirical studies, 62 targeted the C language family and out of the 76 mutation testing tools, only 15 targeted the C language family [23].

Safety-critical systems –where safety standards dictate high statement coverage– are therefore a prime candidate for validating optimisation strategies. In safety-critical software, C and C++ dominate the technology stack [37]. We, therefore, contribute to the mutation testing knowledge as our mutation optimisation techniques target the C langue family, or are language independent.

Proof-of-Concept Tools. We have implemented different proof-of-concept tools in order to perform feasibility analyses and case studies to validate the performance improvements of the optimisation techniques. These proof-of-concept tools are publicly available. Our distribution tool DiMuTesTas is available on GitHub.[5] Our open-source research tool F-ASTMut that implements a variety of mutation optimisation techniques by exploiting the compiler infrastructure, such as the Clang front-end. F-ASTMut is available on Github.[6]

Detailed Measurements. During our work, we provided detailed measurements of the different phases of mutation analyses for both an unoptimised approach as a baseline and for our different optimisation techniques. This allows an in-depth analysis of the potential reduction in compilation and/or execution overhead of the optimisation technique and highlights which parts of the mutation testing phases are prime candidates for optimisations. Finally, the detailed measurements allow us to investigate where potential overheads from these techniques occur, how to mitigate them, and suggest potential avenues for improvements.

Industrial Case Studies. For each of our optimisation techniques, we performed detailed analyses on both open-source projects and industrial projects. The industrial analysis provides additional value as they serve as real-world applications of the evaluated techniques and approaches in professional settings, helping to demonstrate their practicality, relevance, and generalisability. They also offer valuable insights and lessons that cannot be gained from only open-source projects, making them valuable sources of information for practitioners.

Empirical Evaluation. At the end of our work, we did an empirical study on the views on mutation testing in industry. We find this to be a crucial contribution as it provides insight into the level of awareness and understanding of mutation testing amongst professional developers. Our empirical study highlight important information that identifies knowledge gaps or misconceptions that may be hindering the adoption and implementation of mutation testing in industry. Additionally, our empirical study helps to identify the factors that influence the perceived value and utility of mutation testing, such as the costs and benefits of using the technique, the level of training and support required, and the perceived difficulty of implementing it. The knowledge gained from the empirical study can be incorporated into the strategies for promoting the adoption and successful implementation of mutation testing in industry. Finally, our empirical study identifies areas for improvement and potential barriers to the wider adoption of the technique, which can inform future research and development efforts.

[5] `https://github.com/Sten-Vercammen/DiMuTesTas`
[6] `https://github.com/Sten-Vercammen/F-ASTMut`

1.7.2 Research Objectives

Goal 1 – *generate fewer mutants by excluding mutants that do not add value*
To reduce the number of mutants we generate, we utilise the semantic analyser from the Clang front end to prevent the generation of invalid mutants. We also implement a dynamic analysis to detect which mutants are reachable by the test suite. These techniques allow us to exclude the invalid mutants and the completely unreachable mutants, without information loss. We validated this goal in Paper D (p. 69) with four open-source projects and one industrial project from SAAB.

Goal 2 – *execute mutants faster by reducing the execution and compilation overhead*
To reduce the compilation overhead we implemented a mutant schemata technique allowing the compilation of all mutants at once instead of compiling once for each mutant. As all mutants are included in the code base, additional functionality is instrumented to activate a single mutant at a time during runtime. Here we saw a reduction in compilation time to the point that it is only slightly longer than the compilation of the original program. We validated this goal in Paper D (p. 69) with four open-source projects and one industrial project from SAAB. With the compilation overhead virtually eliminated, the execution of the mutants becomes the most time-consuming part. To reduce the execution overhead, we implement a proof-of-concept static analysis method, called goal-oriented mutation testing, that reduces the test suite per mutant to only those test cases that have the responsibility of testing the method in which the mutant resides. This effectively reduces the number of test cases that are executed per mutant to a handful. We saw promising execution speedups of 575x [2]. These speedups are directly linked to the size of the test suite, the more test cases there are the larger the speedup will be. However, additional work needs to be done before the technique can be used in practice, as most importantly, the analysis does not yet support pointers [3].

We then developed an alternative approach by implementing an extension to the mutant schemata approach, called reachable schemata. It has the same goal of reducing the test scope per mutant as the goal-oriented technique, but by utilising dynamic analysis. This technique instruments the code base in order to extract which mutants are actually reached by which test case. Here we saw an average reduction in the test suite size to 10%. We validated this goal in Paper D (p. 69) with four open-source projects and one industrial project from SAAB.

Goal 3 – *process mutants smarter by exploiting the computer infrastructure*
We implemented a split-stream mutation testing strategy that reduces the execution overhead of mutation testing even further. Instead of letting each mutant initiate execution from the start of the program, each mutant is started from the mutation point itself. This is achieved by exploiting the state-space information. By retaining state information between test runs, split-stream mutation testing avoids the redundant execution of statements up until the mutation point.

In theory, this strategy could yield a speedup of a factor 2 to 3, but in practice applying the strategy proves to be too demanding. The split-stream mutation technique demands that dependencies need to be revertible to specific states of the execution. Reverting to the internal state demands too much specific knowledge about the design of the system under test, especially in case data is stored in external databases and filesystems. We validated this approach in Paper D (p. 69) with four open-source projects and one industrial project from SAAB, but we could not eliminate the execution overhead with the split-stream mutation testing strategy.

Finally, we implemented a distribution tool that speeds up the mutation analysis by distributing the workload over a cloud infrastructure. While we validated this using a Java mutation tool on an industrial project from HealthConnect and an industrial project from Intris in Paper A (p. 29), virtually any mutation testing tool can be used in its place.

Goal 4 – *validate mutation testing advances with the intent of industrial adoption*
We performed an empirical study examining the industrial perspective of mutation testing from two companies developing automation and mission safety/critical software in Chapter F (p. 117). The case study research provides relevant viewpoints on the requirements for the industrial adoption of mutation testing techniques. In particular, we obtain new insights concerning the costs vs. benefits of mutation testing, given the potential improvements for computational performance reported in the earlier chapters.

Concerning *benefits*, we observed that mutation testing contributes to a requirements-based testing process, sometimes even revealing flawed or poorly phrased requirements. It can also help with a shift-left testing strategy, as mutation testing reveals under-tested parts of the code early in the CI/CD pipeline. In that respect, our study indicates that mutation testing is not a replacement for human code reviews, but it is a useful support tool for automatic code reviews. Mutation testing identifies and remedies the low-hanging fruits, increasing the overall code quality and thus offloading the human reviewer.

As far as *costs* are concerned, even with the proposed optimisation strategies in place, it remains impractical to aim for killing all mutants due to time constraints. The optimisation strategies we investigated in the earlier chapters are then complementary to other optimisations, such as adopting a limited set of mutation operators or only executing the most relevant mutants. An interesting observation was that equivalent mutants are less of an issue than considered in academic circles and are handled with minimal effort. On the other hand, mutants that cause non-deterministic behaviour, i.e. *flaky mutants*, or mutating already non-deterministic production code, appear to be real concerns that need to be addressed when considering mutation testing in the long run.

The study shows that the mutation testing optimisation techniques from the *do fewer*, *do faster*, and *do smarter* approaches allow the mutation analysis to be integrated into a continuous integration setting. Finally, our study suggests that the industrial perception of mutation testing is evolving as more organisations recognise the potential benefits of the technique and work to address its limitations and challenges.

1.8 Limitations

In order to focus on our research goals where we investigate the potential speedups of the mutation optimisation techniques, we had to impose boundaries on the functionalities of our proof-of-concept tools. While this does limit the capabilities of our research tools, it does not impact the validity of our research. These limitations exist due to the fixed timespan of the Ph.D. thesis and the inherent limitations in development capabilities. We list the limitations below together with their implications and potential solutions.

Goal-oriented. Our goal-oriented mutation technique offers a drastic speedup by limiting the set of test cases that need to be executed for each mutant. As this technique relies on a static analysis of the project, it requires no execution of the test suite in order to establish the traceability links between the test cases and the mutants. We created a proof-of-concept tool that can currently correctly create 77% of the links between the test cases and the mutants with support for private methods. Further research and development is needed to increase its accuracy, as it can currently not be accurately used in "no throw assertion tests" and cannot follow pointers. While the technique cannot yet be used as intended, it can still be utilised as a test execution order optimisation. The goal-oriented technique selects the test cases that are most related to the mutant. These have a higher likelihood of detecting the mutant. Executing these test cases first reduces the time to detect the mutant, effectively speeding up the mutation testing.

Mutation Operator Support. Our open-source research tool F-ASTMut currently supports the Relational Operator Replacement (ROR), Arithmetic Operator Replacement (AOR) and Logical Connector Replacement (LCR). The tool is designed to be easily extendable with other binary mutation operators and unary operators. However, other mutation operator types, like the Access Modifiers Change (AMC) where e.g. the public access label is changed to private, might require substantial implementation effort.

Const, constexpr, and templates. Our driver for mutant schemata relies on information from outside the program to control the activation of the mutants by setting a MUTANT_NR variable. The MUTANT_NR variable is initialised at runtime and will thus never be const. This means that we cannot use the variable inside const and constexpr functions, as these functions are evaluated at compile time and the MUTANT_NR value cannot be known at compile time. Mutating const and constexpr will need to be done differently as non-const functions. This includes type definitions (e.g. *using ...*), template arguments, static_asserts, etc.

For now, we choose not to implement support for these kinds of mutations. They can be implemented by creating separate values and/or functions for each mutated operation and by selecting the correct one everywhere in the project where they are used (e.g. const val becomes const val_0, const val_1, const val_2, ...). The impact of this will need to be investigated as it will drastically increase the size of the code base and the compiled binary.

1.9 Lessons Learned

Throughout the course of this project, we gained valuable insights and lessons about mutation testing. We list them below as they can provide insights and avenues for future research.

Application domain. Mutation testing is primarily intended to assess the *fault-detection capability* of a test suite. However, additional application domains do exist.

Mutation testing can be used as the underlying method to automatically extend an existing test suite with effective new test cases and/or to expand the existing test cases to improve their test effectiveness. We ourselves did an exploratory study in order to reduce the list of survived mutants by automatically detecting the trivial mutants by expanding the existing test cases with simple assert statements [8].

Type Safety. Some challenges occur when we try to implement mutant schemata for statically typed programming languages like C++. First and foremost, the mutated program must be syntactically correct and no type errors should occur. This means that every mutated statement should be valid. We cannot generate mutants like "string − string" or "float % int". Classes can implement or omit operators like "+" and "−" further complicating the matter. Clang allows us to access all the statically available information of the project and to verify if a mutated statement is syntactically correct without the need to compile the complete project.

Instrumentation Overhead. Optimisation techniques that instrument the code base always create a certain overhead when executing the mutants. We envisioned that there would be a fixed overhead cost for the technique per mutant, but that this overhead would remain limited. In our results, we saw that this was only the case for projects with a low to medium number of mutants per line of production code. For projects where the number of mutants per line of production code is high, such as math-heavy projects, our implementation caused an execution overhead that negatively impacts the speedup. For our mutant schemata technique, we utilised *if* statements, but math-heavy projects should benefit from a switch-case implementation. While we point out avenues for improvements, specific research will need to be performed to understand where and how an optimal alternative implementation can be used to achieve the full potential of the mutant schemata approach.

Test per test case vs per module. Our reachable schemata technique extracts the reachable mutants per test case or per module for the program under test. The best results are obtained by extracting the reachable mutants on a test-by-test basis. However, it is possible that the test driver of the test suite cannot execute individual tests but only groups of tests, or so-called test modules. The speedup from this technique is reduced when executing on a per-module basis. Changing the test framework to allow the execution of individual tests might be considered for additional speedup.

Timed Out Mutants Overhead. For the bigger projects we analysed like CppCheck, we have seen that the execution time of the optimised mutation analysis consists mostly of timed-out mutants. Here, the timed-out mutants take up 93% of the reachable schemata mutation analysis. In our current approach, we detected the timeout once the total time for that mutant reached a threshold. For the CppCheck project, this means that a test that is stuck in an infinite loop would only be timed out after 45 seconds. As the CppCheck project has 3,745 test cases, the average test time is below 0.01 seconds. The impact of the timed-out mutants will be drastically reduced if we stop mutants not after a global threshold but after localised thresholds based on the individual tests.

External Dependencies. Many large scale projects have external dependencies like databases. Traditional, unoptimised mutation testing and techniques like mutant schemata are highly likely to support these external dependencies. In Continuous Integration settings, after a test suite has run, the external dependencies, like input/output files and databases, need to be reset before the next run of the Continuous Integration. The support for this is usually baked into the build systems of the projects (e.g. make clean).

Split-stream mutation testing on the other hand requires the external dependencies to be able to be reset to a specific state. While we built in support to reset local files to a specific state using a local GitHub repository, more advanced dependencies like running, or even off-site, databases need very specific commands and domain knowledge to enable the strategy to reset them to a specific state. The implementation for this will be different for each project. This strategy requires too much knowledge of the system. It cannot easily be incorporated with external dependencies, e.g. databases. Our current recommendation is to not use split-stream mutation testing.

Equivalent Mutants. Academic literature indicates that equivalent mutants are a widespread and problematic phenomenon, requiring significant human effort to detect and resolve [41, 62, 63]. However, in our empirical study within industry, we have seen that they can effectively manage the equivalent mutants with minimal effort. By carefully selecting the mutation operators, filtering out non-interesting mutants, and utilising trivial compiler equivalence, they ensure that the testers see almost no equivalent mutants. For the few they do, they can label them as such in the code and avoid future generation of them. As a result, equivalent mutants pose no issues for them adding minimal overhead to the mutation analysis.

Flaky Mutants. An underlying, rarely mentioned, assumption for mutation testing, is that the system under test behaves deterministically. Therefore, we challenged this assumption and investigated the ramifications of mutation testing of non-deterministic code and the introduction of non-determinism due to injected mutants. Testers can only identify non-determinism by confirming the existence of –so-called– flaky tests which intermittently pass and fail whilst executing the same code. This makes testing for non-determinism in source code inherently challenging and impossible to detect from a single execution. This wastes developer time, may hide bugs, and can lead to distrust in the test suit [64–66].

Studies show that flaky tests are a substantial and frequent problem [67–73]. We, however, found no study relating this to mutation testing. For a mutation analysis, many test cases need to be executed for each mutant. This drastically increases the likelihood of flakiness substantially, especially as each mutant also changes the code. It is thus very likely that some mutants introduce non-determinism. We call these *flaky mutants*.

Running a mutation analysis on production code that already contains flakiness makes things even more unpredictable. Introducing a mutant could exaggerate the flakiness, maintain the flakiness, or even fix the flakiness. We currently have no automated way of knowing if a test really detected the mutant or if it failed due to pre-existing flakiness. It is possible that a test fails, causing the mutant to be killed, even though the mutated code was never reached. The test can also fail unrelated to the mutant, even after the mutant was executed.

We investigated open-source software repositories and found many commits dedicated to fixing flaky tests. Of these commits, there were numerous examples where it is possible to go from the committed code, in which the flakiness was solved, to the original code with the use of a mutation operator. This shows that flaky mutants can be generated in real projects. Our empirical study within industry reaffirmed that flaky mutants are an actual problem with mutation testing as they appeared too often such that developers started to complain about them [6].

We presented these findings in the IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2021 [9]. As a first version into tackling the flaky mutants problem, we propose to utilise existing tools that aim to detect flaky tests to detect the flakiness in mutated code [72, 74, 75]. Our mutation testing use case brings an important advantage compared to conventional flaky test management. As we control the location of mutant insertion, we know where the potential root cause of the flakiness lies. Consequently, we can provide this information to the flaky test detection tools. As a starting point, we thus propose to build a model of potentially flaky mutants based on Figure 1.4. The figure represents the taxonomy we constructed from the flaky examples we found in the open-source software repositories and an analysis from undefined behaviour in C/C++ and Java. Our taxonomy roughly aligns with previous studies related to flaky tests and bugs, but with additions and adaptations to cover our identified causes of flaky mutants [65, 70].

Our proposed taxonomy is organised into three main categories. The first, *inserted*, specifies how deterministic source code can turn into a flaky variant. The second category covers *inherited flakiness*, meaning the code is flaky, but we cannot generate simple mutants for them. An example would be switching compilers or test order dependency. The last category includes the *already flaky tests*. For such examples, mutation testing can cause the flakiness to appear more frequently. Examples for each category can be found in our publicly available dataset [76].

Flaky mutants are a real concern that needs to be addressed when considering mutation testing in the long run. Additional research will have to be performed and specialised tools will need to be created.

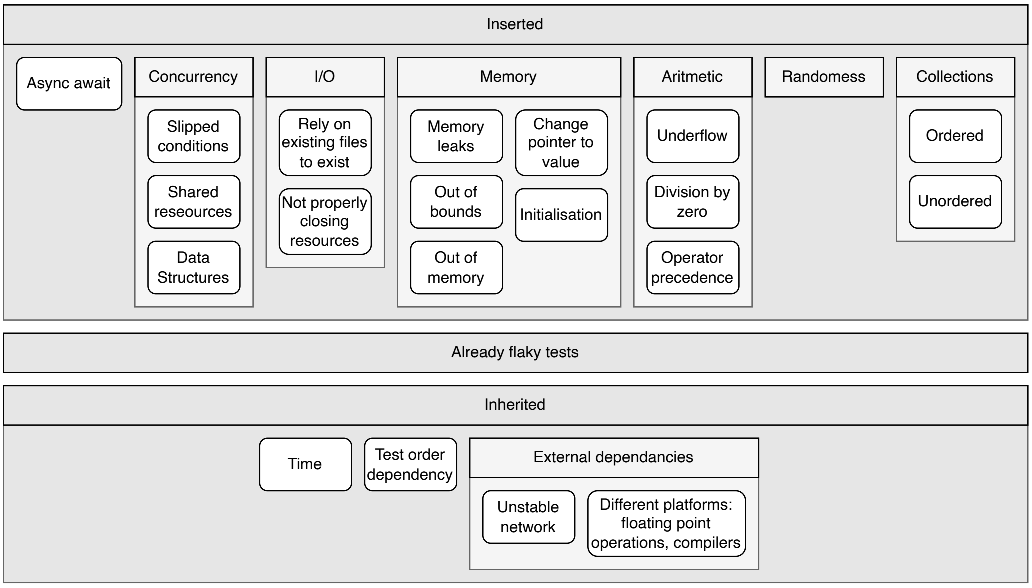


Figure 1.4: Causes of Flaky Mutants

1.10 Future Work

Each of our optimisation techniques orthogonally speed up the mutation testing analysis. However they have been selected and designed in order to work together. Creating a mutation testing tool that includes all of the above mentioned optimisations would deliver an even greater speedup. Naturally, there are additional improvements that can be applied to the existing techniques. One could optimise the schemata implementation to offset the overhead introduced in mathematical heavy libraries. One could also investigate how to integrate the equivalent mutant detections techniques with the other mutation testing optimisations.

1.11 Conclusion

To make mutation testing effective in an industrial setting, we set three objectives: (1) generate *fewer* mutants, (2) execute them *faster* and (3) process them *smarter*. For each of these objectives, we investigated the most promising techniques from the current state-of-the-art and implement them into a proof-of-concept tool.

Our prime candidates for mutation testing are safety-critical systems, where C and C++ dominate the technology stack [37]. Yet this is not represented in the mutation testing community as less than 20% of the created mutation testing tools target the C language family [23]. Therefore, we tailored our research and optimisation techniques to the C language family.

For each of our optimisation techniques, we performed detailed analysis on both open-source projects and industrial projects. At the end of our work, we performed an empirical study on the views on mutation testing in industry, to evaluate whether the advances of the mutation testing optimisations are sufficient for industrial adoption and to identify areas for improvement and the potential barriers that prevent the wider adoption of mutation testing in industry.

Goal 1 – generate *fewer* mutants by excluding mutants that do not add value
We demonstrated the feasibility of using the Clang compiler front-end for different optimisation strategies and evaluated this on four open-source projects and one industrial project from SAAB. We have shown that we can leverage the semantic analyser of the Clang front-end to ensure, during the mutant generation phase, that the created mutants are compile-time correct. While this reduces the number of mutants that we need to analyse, it is a requirement for a mutant schemata based optimisation. We also implemented a technique to exclude the mutants that are not reachable by the test suite. This speedup, however, depends on the actual coverage of the test suite: a test suite with high coverage (thus reaching more mutants) yields a lower speedup.

Goal 2 – execute mutants *faster* by reducing the execution and compilation overhead
With the reachable schemata strategy, we virtually eliminated the compilation overhead to the point that the compilation for all mutants was only slightly longer compared to the original compilation time and we reduced the execution overhead by only executing the test cases for individual mutants which actually reach said mutants. Compared to an *unoptimised* approach we achieve a maximum speedup of 23.45x and 30.52x on the JSON and Google Test projects with the *reachable schemata* strategy. Even for less ideal scenarios from the CPPCheck and TinyXML2 projects we achieve a speedup of 2.07x and 5.89x. These can be sped up further by two optimisations: Firstly, use *switch* statements for the mutant selection to reduce the technique overhead. Secondly, tailoring the time-out function, that e.g. detects mutants stuck in infinite loops, on a test-by-test basis instead of on the global test suite. Our most important lesson learned is that we need a different, specialised approach for generating mutants in *const*, *constexpr*, *templates*, and *define* macros. These statements are evaluated at compile-time, thus obstructing the runtime selection of the mutant required by the mutant schemata technique.

With four open-source projects and one industrial project from SAAB, we have shown that the reachable schemata technique reduces the test suite scope to approximately 10% for each mutant. This can be further reduced by the goal-oriented mutation technique to only those test cases that have the responsibility of testing the method in which the mutant is located. However, some extensions are needed to improve the accuracy of the tool. Most importantly, the analysis needs to be able to deal with "no throw assertion tests" and follow pointers.

Goal 3 – process mutants *smarter* by exploiting the computer infrastructure
Our proof-of-concept implementation of a split-stream mutation testing strategy shows that the execution overhead of mutation testing can be further reduced. Instead of letting each mutant initiate execution from the start of the program, each mutant is started from the mutation point itself. This is achieved by exploiting the state-space information. By retaining state information between test runs, split-stream mutation testing avoids the redundant execution of statements up until the mutation point.

In theory, this strategy could yield a speedup of a factor 2 to 3, but in practice applying the strategy proves to be too demanding. The split-stream mutation technique demands that dependencies need to be revertible to specific states of the execution. Reverting to the internal state demands too much specific knowledge about the design of the system under test, especially in the case data is stored in external databases and filesystems. Our current recommendation is to not use the strategy.

Finally, we implemented a cloud solution to distribute the mutation workload over a cloud infrastructure. Here we saw that by doubling the number of hardware nodes, the execution time almost halves, nearly increasing the speed-up linearly. Despite these improvements, there are still opportunities for further optimisation. We pointed out directions for future work based on detailed measurements concerning delays in the analysis. In particular, the use of *multicast* should ensure that the *set-up delay* —the current bottleneck– would take constant time, regardless of the number of nodes in the system. In the same vein, we can minimise the *file server/disk delay* by only sending the deltas of the files. Our most important lesson learned is that tasks should remain completely independent for optimal deployment in a cloud infrastructure. Mutant optimisation techniques that violate this principle benefit less from deploying in the cloud. Nevertheless, there is ample room for complementary optimisation techniques that reduce the time needed to generate and execute mutants.

Goal 4 – validate mutation testing advances with the intent of industrial adoption
We performed an empirical study examining the industrial perspective of mutation testing with two companies developing automation and mission safety/critical software. As part of this goal, we explored whether mutation testing provides sufficient benefits for industrial adoption and whether the recent mutation testing advances are sufficient to meet the computational performance requirements for industrial adoption. For this, we integrated our optimised mutation testing tool in the company developing automation software, and performed the mutation analysis on a multitude of their software components. To contrast their perspective, we were also in contact with a company developing mission safety/critical software that had 5 years of experience with mutation testing and fully integrated it within one of their development teams.

In our study, we learned that the initial perspective of mutation testing is that they need to kill all mutants, which due to time constraints, is impractical for large-scale applications. However, in practice a trade-off is made by using a limited set of mutation operators and only executing the most-relevant mutants. An unintended side effect of this approach is that very few equivalent mutants are being generated, most of which are caught by utilising a trivial compiler equivalence technique. This causes equivalent mutants to be less of an issue than considered in academic circles and shows that they can be handled with minimal effort. On the other hand, mutants that cause non-deterministic behaviour, i.e. *flaky mutants*, or mutating already non-deterministic production code, appear to be real concerns that need to be addressed when considering mutation testing in the long run. While we previously investigated the flaky mutant phenomenon, and proposed a first vision on how to deal with them, seeing them occur in industry practice only validates that they are a real concern and that they need to be addressed. For this, additional research and tools is needed.

Our study shows that the mutation testing optimisation techniques from the *do fewer*, *do faster*, and *do smarter* approaches allow the mutation analysis to be integrated into a continuous integration setting. However, mutation testing is not used as a replacement for human code reviews. It is a useful tool for automatic code reviews, identifying and correcting the low-hanging fruits, increasing the overall code quality, and reducing the workload of the human reviewer.

Overall, our study suggests that the industrial perception of mutation testing is evolving as more organisations recognise the potential benefits of the technique and work to address its limitations and challenges. Mutation testing helps with a shift-left testing strategy, and contributes to a requirements-based testing process, sometimes even revealing flawed or poorly phrased requirements.

Concluding Thesis Statement — The mutation optimisation techniques investigated for the C language family, following the (1) generate *fewer* mutants, (2) execute them *faster*, and (3) process them *smarter* approach, allows for mutation testing to be adopted in a continuous integration setting for safety-critical systems. However, due to time constraints, killing all mutants is impractical for large-scale applications. In practice, a middle ground is advisable where mutation operators are selected carefully, and only the mutants that are the most relevant or interesting to the developers are executed. This allows the mutation testing to help with a shift-left testing strategy, without overwhelming developers. While equivalent mutants occur, they appear to be less of an issue than previously acknowledged in academic circles. Instead, they might be handled with minimal effort in practice. This is in contrast to flaky mutants which introduce non-deterministic behaviour, as they appear in open-source projects and industry. They are a real concern that needs to be addressed when considering mutation testing in the long run.

Publications

Paper A

Speeding up Mutation Testing via the Cloud: Lessons Learned for Further Optimisations

Sten Vercammen
University of Antwerp, Belgium
Antwerp, Belgium

Serge Demeyer
University of Antwerp, Belgium
Antwerp, Belgium

Markus Borg
RISE SICS AB
Lund, Sweden

Sigrid Eldh
Ericsson AB
Stockholm, Sweden

Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2018

Abstract

Background: Mutation testing is the state-of-the-art technique for assessing the fault detection capacity of a test suite. Unfortunately, it is seldom applied in practice because it is computationally expensive. We witnessed 48 hours of mutation testing time on a test suite comprising 272 unit tests and 5,258 lines of test code for testing a project with 48,873 lines of production code. **Aims:** Therefore, researchers are currently investigating cloud solutions, hoping to achieve sufficient speed-up to allow for a complete mutation test run during the nightly build. **Method:** In this paper we evaluate mutation testing in the cloud against two industrial projects. **Results:** With our proof-of-concept, we achieved a speed-up between 12x and 12.7x on a cloud infrastructure with 16 nodes. This allowed to reduce the aforementioned 48 hours of mutation testing time to 3.7 hours. **Conclusions:** We make a detailed analysis of the delays induced by the distributed architecture, point out avenues for further optimisation and elaborate on the lessons learned for the mutation testing community. Most importantly, we learned that for optimal deployment in a cloud infrastructure, tasks should remain completely independent. Mutant optimisation techniques that violate this principle will benefit less from deploying in the cloud.

A.1 Introduction

Software testing is the dominant method for quality assurance and quality control in software development organisations [10, 11]. Software testing was established as a disciplined approach in the late 70's when it was defined as *"executing a program with the intent of finding an error"* [12]. In the last decade, this intent shifted dramatically with the advent of continuous integration [13]. Many software tests are now fully automated, and serve as quality gates, safeguarding against programming faults. The scale at which automated software tests are adopted in modern software organisations is mind-boggling. Microsoft for instance reported that approximately 11 months of development on Windows comprised more than 30 million test executions. Google on the other hand reported that *"In an average day, TAP integrates and tests [...] more than 13K code projects, requiring 800K builds and 150 Million test runs."* [14]. As a result of this continuous integration approach, software organisations are capable of releasing faster. Tesla, for example uploads new software in its cars once every month [15]. Amazon pushes new updates to production every 11.6 seconds [16].

The growing reliance on automated software tests raises a fundamental question: How trustworthy are these automated tests? Today, mutation testing is the state-of-the-art technique for assessing the *fault-detection capacity* of a test suite [22]. The technique deliberately injects faults into the system under test and counts how many of them are caught by the test suite. Mutation testing is acknowledged within academic circles as the most promising technique for a fully automated assessment of the strength of a test suite [23]. One of the reasons mutation testing is seldom adopted in industrial settings is because the technique is computationally expensive: each mutant must be deployed and tested separately [22].

Mutation testing has shown to be able to run in parallel on a distributed architecture [46, 48, 49]. Researchers are currently investigating cloud solutions to share the computational load across a series of hardware nodes. Most notably among them are Hadoopmutator [50] and Eminent [51]. These tool prototypes demonstrate that cloud infrastructure indeed allows to speed up the mutation testing. Yet, today it is unclear how to optimally distribute the load across the available hardware nodes.

In this paper we evaluate an alternative cloud solution (named DiMuTesTas) against two industrial projects, one small and one large. We achieve a speed-up between 12x and 12.7x on a cloud infrastructure with 16 workers, illustrating that substantial speed-up is possible yet that the overhead is significant. We collect detailed measurements on the cloud infrastructure (setup, scheduling, file transfer), analyse how the overhead occurs, suggest avenues for further improvements and elaborate on the lessons learned for the mutation testing community.

The rest of the paper is structured as follows. In Section A.2, we elaborate on the concept of mutation testing and list related work. In Section A.3, we describe the cloud architecture of our proof-of-concept, identifying where to measure overhead. In Section A.4, we explain our case study setup, which naturally leads to Section A.5 where we discuss the results. In Section A.6 we derive the lessons learned. As with any empirical research, we list the threats to validity in Section A.7 to arrive at a conclusion in Section A.8.

A.2 Background and Related Work

In this section, we elaborate on the concept of mutation testing and contrast our proof-of-concept against related work.

A.2.1 Mutation Testing

For effective testing, software teams need strong tests which maximise the likelihood of exposing faults [12]. Traditionally, the strength of a test suite is assessed using code coverage, revealing which statements are poorly tested. However, code coverage has been shown to be a poor indicator of test effectiveness [17, 19]. Worse, even a 100% MC/DC coverage (Modified Condition/Decision Coverage, the coverage criterion adopted for safety critical systems) still does not guarantee the absence of faults [20, 21].

Today, mutation testing is the state-of-the-art technique for assessing the *fault-detection capacity* of a test suite [22, 23]. The technique deliberately injects faults (called mutants) into the production code and counts how many of them are caught by the test suite. Case studies with safety critical systems demonstrate that mutation testing could be effective where traditional structural coverage analysis and manual peer review have failed [33, 34]. Google on the other hand reports that mutation testing provides insight into poorly tested parts of the system, but –more importantly– also reveals design problems with components that are difficult to test, hence must be refactored [35].

Unfortunately, mutation testing is seldom adopted in practice [77]. One of the reasons is that the technique —in its basic form– requires a tremendous amount of computing power. For each injected mutant, the code base must be compiled and tested separately [22]. Algorithm A.1 shows the essential steps without any optimisations, in order to understand the time-consuming nature of the mutation testing process. The software system needs to build without errors and all software tests should succeed before mutation testing can even begin; this is called the *pre*-phase. Then, the two main phases are executed: (A) the mutant generation phase and (B) the mutant execution phase. In phase A, mutants are generated for all source files. In phase B, each mutant is executed and its result (whether or not it was killed) is saved. Finally, all the results are gathered and the final report is created in the *post*-phase.

Algorithm A.1 Pseudocode Mutation Testing

```
1:  function MUTATIONTESTING(srcFolder $src$)
2:      ▷ Pre: verify build and if all tests succeed
3:      if INITIALBUILDANDTESTS() $\neq$ True then
4:          return
5:
6:      ▷ A: generate mutants
7:      $mutants \leftarrow []$
8:      for all srcFile $f \in src$ do
9:          $fMutants \leftarrow$ GENERATEMUTANTS(srcFile $f$)
10:         $mutants \leftarrow mutants + fMutants$
11:
12:     ▷ B: execute mutants
13:     for all mutant $m \in mutants$ do
14:         $result \leftarrow$ EXECUTEMUTANT(mutant $m$)
15:         STORERESULT($result$, mutant $m$)
16:
17:     ▷ Post: process results
18:     PROCESSRESULTS()
```

A.2.2 Mutation Testing Optimisations

A lot of research is devoted to optimising the mutation testing process, summarised under the vision - *do fewer*, *do smarter*, and *do faster* [42]. The *do fewer* approaches minimise the execution time by reducing the total number of mutants to execute. Such an optimisation can be implemented by generating fewer mutants on line 9 in Algorithm A.1 or by selecting a subset of all mutants on line 13. The fewer mutants that are executed, the more information will be lost. Balancing time reduction versus information loss is key. There are different ways to choose which mutants will be executed, varying in their effectiveness compared to the full set of mutants [22]. *Do smarter* approaches attempt to minimise the execution time by retaining state information between runs, e.g. split-stream mutation testing [47]. Others *prioritise test*, giving priority to the tests with the highest likelihood of failure. These optimisations would be implemented on line 14 in Algorithm A.1. Lastly, *do faster* approaches try to minimise the execution time of each individual mutant. One example is using a compiler integrated technique, where the project is compiled only once instead of for each mutant [44]. These optimisations would also be implemented on line 14 in Algorithm A.1.

A.2.3 Mutation Testing in the Cloud

Mutation testing has shown to be able to run in parallel on a distributed architecture [46, 48, 49]. Consequently, researchers are currently investigating cloud solutions to share the computational load across a series of hardware nodes. Hadoopmutator [50] and Eminent [51] are the ones we have found in the literature.

Hadoopmutator

Hadoopmutator is a cloud-based Mutation Testing framework that is implemented using Hadoop's MapReduce[1]. During the mapping phase, each mutation operator is assigned to a separate node, creating a single mutant and executing the test suite. The subsequent reduce phase aggregates the results from all the test executions and calculates the mutation score.

Data transfer between the nodes is handled using Hadoop's MapReduce and the Hadoop Distributed File System (HDFS™). Hadoopmutator is applied on two open source projects, and the authors report a speed-up of 9.57x and 12.83x using 13 nodes. For small projects like Apache Wicket, the authors state that "*the overhead of running Hadoop on the compute nodes becomes significant relative to the time needed to generate and executes the tests and hence the optimal performance gain is not attained*". The influence of speed-up with large projects that affect the I/O and network traffic was not investigated.

For a MapReduce solution to reach an optimal load balance, there should be small variance between the execution times of the respective mapper functions. This minimises the inherent idle time between the termination of the mapper phase and the start of the reduce phase. However, for mutation testing this principle does not hold. First, because the analysis can terminate as soon as one test fails, thus some test executions will terminate earlier than others. Second, because some mutants lead to infinite loops, which can only be detected via time-outs. Therefore there is a large variance between the time to execute the tests.

Eminent

Eminent (EMbarrassINgly parallEl mutatioN Testing) is a distributed Mutation Testing tool that relies on the Message Passing Interface standard for portable message-passing in parallel computing architectures [51]. Eminent uses *test-level granularity*, the test suite is split up and each test/mutant combination is run separately. If a single test (of a mutant) fails, the mutant is killed and all other remaining tests (related to that mutant) are canceled.

Eminent handles data transfer between the master and the nodes by means of a shared database. The test cases are sent to the worker processes which will execute them against the mutants and send the results back to the master, which compares them to the original. Eminent is applied on three projects, and the authors report a speed-up between 8x and 22x using 32 nodes. Large projects can have an impact on the scalability "*because of the high volume of network and I/O traffic generated by this application, which acts as a system bottleneck in the database node*".

[1]`https://hadoop.apache.org`

The main advantage of Eminent's test-level granularity is that the sooner a mutant is killed the faster the mutation testing tool becomes. Therefore, the test execution framework should be configured such as to stop the execution of the test suite when the first one fails. The second advantage of test-level granularity is that the load can in principle be evenly distributed over multiple nodes. Nevertheless, the worst case scenario for test-level granularity occurs when the last test is executed on one node while all the other nodes are finished. The additional overhead of running the test-level granularity may be more than the execution time of the complete test suite.

At the time of writing there was no implementation available for HadoopMutater nor for Eminent. Replicating their results on other projects and measuring where overhead occurred was impossible. Therefore we resorted to our own proof-of-concept named DiMuTesTas.

DiMuTesTas

We developed DiMuTesTas to minimise the idle-time of the nodes and to minimise the network load of the distributed application itself. The first is tackled by removing dependencies between executions on the workers, allowing to distribute the generation of the mutants and the execution of the mutants independently of each other. We keep the network load low by only sending references to files over the network. We use *mutant-level granularity*, thus apply a mutant and execute the complete test suite. Mutant-level granularity allows to further reduce the amount of messages that need to be exchanged.

In our current system, we use a file system to distribute the project, mutants, and store their executed results. The data that needs to be transferred for the mutant is limited to a single file. This can even further be reduced by only sending the delta of the file. Writing to the file server is only done by each worker which has generated the mutants (single files) or by each worker which executed a mutant and needs to store its build output (multiple files).

Summary. The current state-of-the-art demonstrates that cloud infrastructure indeed speeds up the mutation testing. Yet, the optimal way of distributing the load across the available hardware nodes is currently unknown. First of all, there is the potential for idle-time when nodes are waiting for others to finish their tasks before they can proceed. Secondly, there is the data-transfer bottleneck, the consequence of copying files and exchanges messages across the nodes. Today, detailed measurements on the impact of both the idle-time and the data-transfer are lacking.

A.3 Proof of Concept

In this section we first describe the cloud architecture of our proof-of-concept (loosely inspired by the 4+1 model [78]) and then identify where delays may have a significant impact, thus where we should measure overhead.

A.3.1 DiMuTesTas Architecture

Logical View — Single Task queue

To execute Algorithm A.1 in the cloud, we adopt a single task queue model for the main phases *A* and *B*, as depicted in Figure A.1. For phase *A*, this means creating a first kind of task, i.e. to *generate the mutants from a source (src) file* (represented by *A* in Figure A.1), for each source file and push it onto the task queue. When a worker processes such a task (e.g. *1*), a new, second kind of task is created for each of the generated mutants, i.e. to *execute the mutant* (corresponding to phase *B*). These newly created tasks are then pushed back onto the task queue (e.g. *1a*, *1b* and *1c*).

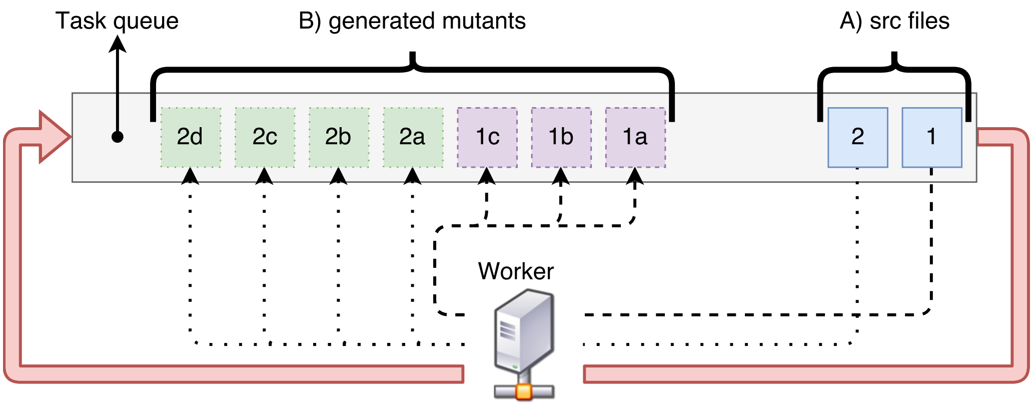


Figure A.1: DiMuTesTas Architecture – Single Task Queue

Process View — RabbitMQ

The single task queue is handled by RabbitMQ[2], a broker which handles the message passing between multiple computers. To map the logical view onto RabbitMQ we follow a series of steps depicted in Figure A.2. First, the master performs the initial build and verifies if all tests succeed (*1*, a.k.a. *pre*-phase); if not, the process is canceled. Afterwards, all source files are gathered and (their file names) are sent to the task queue (*2*). From here on, the master waits until he received all results. Once tasks are in the task queue, workers will pull and process them. If a worker pulls a task containing the name of a source file (*3a*), it will generate mutants for the corresponding file, store them on the file server (*3b*) and send a reference for each mutant back to the queue. If the worker pulls a task containing a reference to a mutant (*4a*), it will fetch the mutant from the file server (*4b*) and execute it before storing its result on the file server (*4c*) and sending a “done”

[2]`https://www.rabbitmq.com/`

message to the result queue (*4d*). Finally, when the master received all results, he creates the final report for the mutation testing (*5*, a.k.a. *post*-phase).

Note that the mutants can be generated and executed by different workers and are needed for the final report as well. Therefore, they are not stored in the task queue as this would increase its memory usage and would require sending the mutants over the network one additional time. Instead, the mutants, results and the final report are fetched stored on a separate file server (*3b*, *3c* and *4c*) .

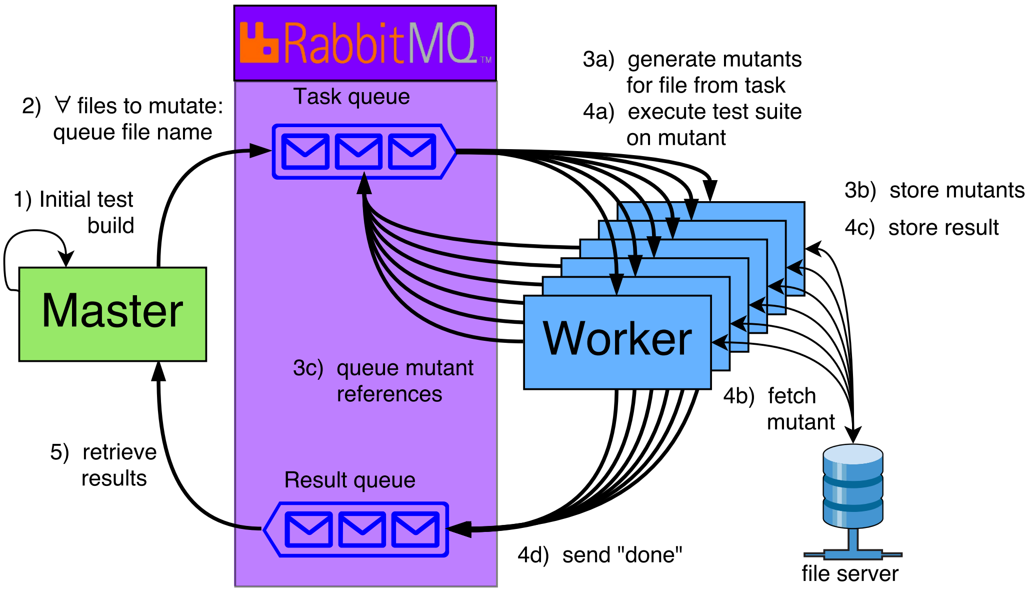


Figure A.2: DiMuTesTas Architecture – Process View

Physical View – Docker

To deploy the architecture on a physical system, we rely on Docker[3]. Figure A.3 provides an overview of the different components and their interactions. As setting up each node individually is impractical, the master and the worker are encapsulated into Docker images, i.e. an executable package that includes everything needed to run an application: the code, the runtime environment, the libraries, the environment variables, and the configuration files. As Docker images contain the installed software, they will startup very quickly because no further installation is required. An image for the RabbitMQ server already existed. The master and worker images do not include the project itself, but will copy it from a file server once they startup. We used NFS for the file server, but iSCSI, FC, and others are possible as well. The file server is also needed as the local storage of each Docker container is removed together with the container after execution. To distribute the tool over multiple PC's and enable the different containers to talk to each other, we make use of Docker Swarm.

[3]`https://docs.docker.com/`

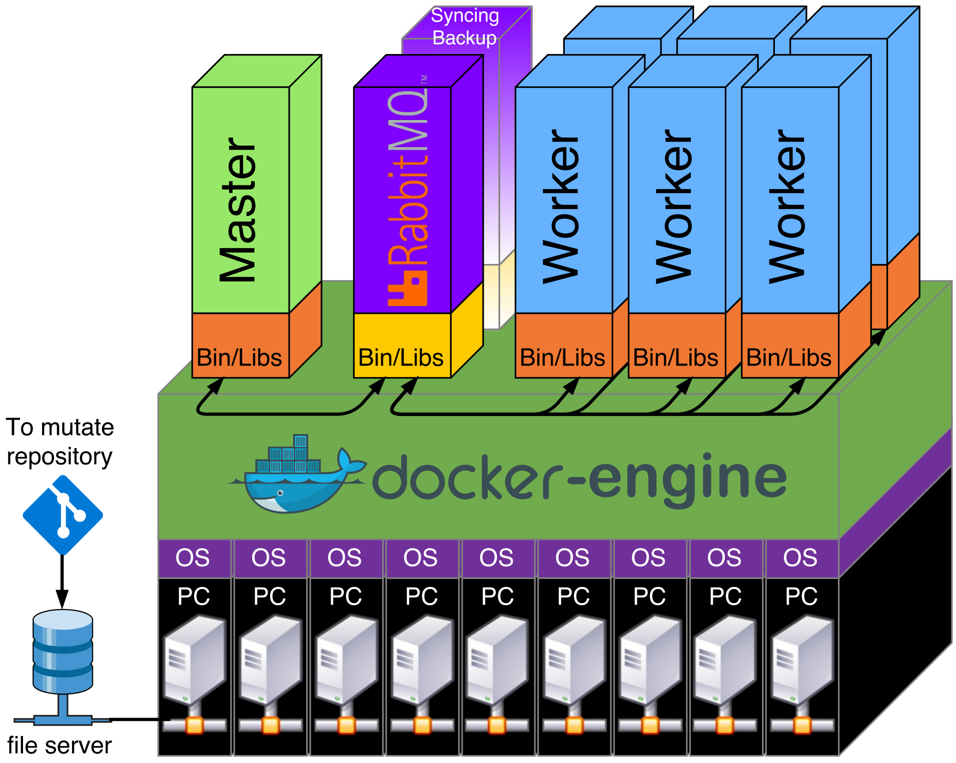


Figure A.3: DiMuTesTas Architecture – Physical View

A.3.2 DiMuTesTas Potential Delays

Now that we laid out the components and how they interact (see Figure A.3) we can identify where potential delays might occur compared to a mutation test run on a local PC.

- *Setup delay.* After the initial build on the master, DiMuTesTas copies the build dependencies from the file server to the own local storage in each worker. This is done to prevent the network connection from becoming a bottleneck, as otherwise each worker has to download the build dependencies separately.
- *Initial build.* The *pre-* and *post*-phase are similar for a mutation test run on a local PC and one that runs in the cloud, differing only in the place they store and gather information.
- *Mutant generation.* The total time all workers needs to generate the mutants, excluding the time to read from the file server/disk and writing the build output to the file server/disk. For a mutation test run on a local machine, the total time to generate the mutants, excluding the time to read or write the results.
- *Mutant execution.* Similar to the *mutant generation* phase, it measures the total time for the mutant execution phase excluding the time to read or write.
- *RabbitMQ (scheduling) delay.* The time to gather the source files and push them to the task queue on the RabbitMQ server. The time needed by the workers to pull

tasks from the task queue is also part of this delay.

- *File server/disk delay.* The most likely cause for delays in a cloud solution is copying data files back and forth between the different nodes. This delay occurs when a container copies the project from the file server to its own local storage. However, it also occurs when transforming Docker images (static, unchangeable) into Docker containers (dynamic, changeable) at startup.

A.4 Case Study Set Up

This section describes how we evaluate the impact of running mutation testing in the cloud. Essentially we apply DiMuTesTas on two industrial projects, comparing a cloud solution against a version running on a local PC.

A.4.1 Cases

Table A.1: Industrial Cases: Descriptive Statistics

Company	Project Start Date	Nr of Commits	Nr of Developers	Java Files	LOPC	LOTC	Test Cases	Branch Coverage
Intris	26 May 2014	27,034	14	85 of 4,070	8,389	343	45	1.27 %
Health-Connect	18 Jun 2014	22,956	10	601 of 273,722	48,873	5,258	272	28.34 %

LPOC = Lines of Production Code; LOTC = Lines of Test Code

Case	Project Size	Test Suite Run Time	build-Output-FileSize	avg-FileSize	interNode-Link-Speed	readWrite-Speed-LocalDisk
Intris	116 MB	7 s	37 kB	15 kB	100 Mbps	95 MBps
HealthConnect	1.8 GB	47 s	840 kB	8.52 kB	100 Mbps	95 MBps

Table A.2: Industrial Cases: Mutation Testing Data

Company	Project Size	Executed Mutants	Invalid Mutants	Actual Mutants	Killed Mutants	Mutation Coverage
Intris	116 MB	1,364	312	1,052	33	3.14 %
HealthConnect	1.8 GB	4,104	50	4,054	360	8.88 %

We collected two cases via our network of industrial partners, one small (Intris), and one large (HealthConnect) project.

- *Case 1: Intris [`https://www.intris.be`].* The project at Intris relies heavily upon the visualisation and manipulation of database data. This manifests itself in the way the tests are written, as most of them are scenario tests. We choose a (core) subproject which does not rely on the database, but has few unit tests. This resulted in a small task execution time, as building the project and running the test suite only takes 7 seconds. The Intris project makes an interesting case for investigating

mutation testing in the cloud as the execution times of the tasks are quite small thus we expect more overhead from scheduling delays and file server/disk delays.
- *Case 2: HealthConnect [`https://www.healthconnect.be`].* The project at HealthConnect is 1.8GB large, and contains 48k LOPC (Lines of Production Code), and 5k LOTC (Lines of Test Code). As each hardware node needs to copy the entire code base, the fileserver hosting the source files from the project may become a bottleneck. The project from HealthConnect makes an interesting case to examine the behaviour on large (especially in data transfer) projects.

The descriptive statistics of the project are listed in Table A.1, while the details regarding basic mutation testing are shown in Table A.2. Note that the given start date and number of commits from HealthConnect is counted from the switch to the new version control system, the actual start date is earlier. Note as well that the time to execute the test suite assumes that all dependencies are loaded and stored locally.

A.4.2 Research Questions

The case study is driven by the following two research questions.

RQ1: Speed-up

How much speed-up can be achieved by running a mutation testing on cloud infrastructure?

Motivation. Assuming that we distribute the mutation test run over a system with N hardware nodes, the ideal speed-up is a factor Nx. Here we investigate whether DiMuTesTas approaches this ideal.

Approach. We deploy DiMuTesTas on a cloud system with a maximum of eight hardware nodes, where each hardware node is configured with 2 workers. We measure the total execution time with a set-up of 1, 2, 4, 8, and 16 workers and average the execution time across 3 runs.

RQ2: Delays

Where does a cloud solution like DiMuTesTas suffer from delays? Do these delays correspond to what may be expected?

Motivation. Deploying mutation testing in the cloud induces delays, in particular with respect to data-transfer between the nodes. This research question compares a mutation test run executed on a local PC against a mutation test run deployed in the cloud. By making a thorough analysis of where delays occur we can suggest avenues for further improvement.

Approach. Based on the set-up described in RQ1 (1, 2, 4, 8, and 16 workers) we measure the points in the architecture where delays might occur as described in Section A.3.2: Setup delay, Initial build, Mutant generation, Mutant execution, RabbitMQ (scheduling) delay and the File server/disk delay. We compare the actual measurements against what we expect.

A.4.3 Hardware Set-up

The infrastructure used for the analysis of both projects is the same. We used 8 Intel(R) Core(TM)2 Quad Q9650 CPU's, each with two 4 GB (Samsung M378B5273DH0-CH9) DDR3 RAM modules and a 250 GB Western Digital (WDC WD2500AAKX-7) hard drive. All PC's were connected to the same subnet using a 3Com Baseline Switch 2016 (100 Mbps, full duplex). The in- and outgoing internet connections/inter-PC communications were limited by the 100 Mb links. We used a dedicated switch to remove any external influences on the network load. All PC's where running Ubuntu 16.04.2 LTS with kernel 4.10.0-27.

The workers from the DiMuTesTas approach where divided equally over the nodes: when using N nodes and $2N$ workers, each node will run two workers. When executing LittleDarwin, only one of the PC's was used.

A.4.4 LittleDarwin

In principle, DiMuTesTas can be set-up with any mutation tool, as long as it can be configured to apply a single mutator on a given file. For this particular case study we relied on LittleDarwin, a tool distributed within our lab thus conveniently accessible [79].

Table A.3: Results (Distributed) Mutation Testing Experiment

	DiMuTesTas				
LittleDarwin	1 worker (1 node)	2 workers (2 nodes)	4 workers (4 nodes)	8 workers (8 nodes)	16 worker (8 nodes)
Intris:					
9,718.15 s	10,360.64 s	5,237.88 s	2,676.80 s	1,404.76 s	810.49 s
93.80 %	100.00 %	50.56 %	25.84 %	13.56 %	7.82 %
HealthConnect:					
173,061.51 s	190,313.06 s	96,218.86 s	48,805.47 s	25,301.98 s	13,618.04 s
90.94 %	100.00 %	50.56 %	25.64 %	13.29 %	7.16 %

A.5 Results

A.5.1 RQ1 - Speed-up

How much speed-up can be achieved by running a mutation testing on cloud infrastructure?

Table A.3 compares the execution times of LittleDarwin against DiMuTesTas using 1, 2, 4, 8, and 16 workers. Each result is an average of 3 runs; the variance between each run is less than 2%. For easy interpretation of the scaling, we took the execution time of DiMuTesTas running on a single worker (therefore running sequentially) as our baseline, indicating it as 100% of the execution time. As the number of workers double, we see that the execution time almost halves thus the speed-up increases linearly. Nevertheless, the relative execution time of LittleDarwin is below 100%, because —as expected— executing mutation testing in the cloud adds overhead.

For the Intris Case (8 kLOC production code, 300 LOC test code) we could reduce the full mutation testing cycle from 2.7 hours to 13.5 minutes. For the HealthConnect case (48 kLOC production code, 5k LOC test code) we could reduce from 48 hours to 3.7 hours. As such, our proof-of-concept achieved a speed-up between 12x and 12.7x on a cloud infrastructure with 16 nodes.

A.5.2 RQ2 - Delay

Where does a cloud solution like DiMuTesTas suffer from delays? Do these delays correspond to what may be expected?

Table A.4: Results: Delays incurred in DiMuTesTas

Program section	Setup delay	Initial build	**Generate Mutants**	**Execute Mutants**	**Rabbit-MQ delay**	**File Server /disk delay**
Intris:						
LittleDarwin	N.A.	7.98	928.46	8,770.34	N.A.	2.58
1 worker	18.39	7.72	1,018.49	9,296.29	9.09	5.24
2 workers	27.83	7.92	558.14	9,791.96	10.69	. 12.94
4 workers	46.63	7.82	332.04	10,090.61	23.71	18.20
8 workers	84.34	8.18	256.38	10,142.68	37.43	19.13
16 workers	*81.59*	8.35	255.23	11,057.54	95.08	21.56
HealthConnect:						
LittleDarwin	N.A.	114.78	253.27	172,681.67	N.A.	6.68
1 worker	156.37	159.03	296.11	189,488.88	22.69	144.86
2 workers	305.91	163.5	257.19	190,937.38	30.73	160.63
4 workers	613.83	177.63	270.29	191,310,10	106.06	187.34
8 workers	1,265.15	165.25	311.63	189,881.46	203.18	190.85
16 workers	*1,319.37*	180.31	269.92	191,940.89	386.73	253.95

(in seconds, **bold** program section represent cumulative timings, i.e. the sum of time all workers spend in that phase)

Table A.5: Overview of Expected and Actual Delays

Program section	Expected delay	Actual delay
Setup delay	Linear to nr. of workers	Linear to nr. of **nodes**
Initial build	Constant	
Mutant generation	Constant	**Decreasing**
Mutant execution	Constant	
RabbitMQ delay	max $n-1$ workers * test execution time	
File server/disk delay	Linear to nr. of workers	

The delays of the different phases are listed in Table A.4. Table A.5 summarises these results, comparing the delay we expected against the delays observed.

Setup delay

In Table A.4, we see that the *setup delay* grows linearly with the number of workers. This is as expected, as the source files from the project are copied to each worker individually.

While we would expect the initial setup delay of the 16 workers to be twice as long as the one of the 8 workers, we see that they are alike. When running the 16 workers, each node has two workers. Because we used NFS as our file server, the kernel caches the data from the requests, allowing the second worker on each node to use the cached data instead of copying it from the file server.

The linear growth of the *setup delay* mainly impacts the execution time of the project from HealthConnect. The larger the project, the more time the copying of the files will take. In our case, the limited network connection of 100Mbps is making the delay extra apparent. The delay can be decreased by using a gigabit network and by minimising the amount of data that needs to be sent over the network. The latter can be achieved by keeping the project in a Docker volume between consecutive runs of the distributed mutation testing.

Another optimisation is to replace unicast with multicast [80]. With unicast, the project is transmitted to each node individually, thus we send the same project 8 times over the network when using 8 nodes. Multicast, on the other hand, sends the (same) project only once over the network, making the amount of data sent over the network independent of the number of nodes.

Initial build

Next in Table A.4 is the *initial build* on the master. We see that the execution times of the initial build from the different workers are similar. As this step is the same, independent of the number of workers, we expected a constant *initial build* delay, hence this result is as expected.

Mutant generation

The third row in Table A.4 shows the total time to *mutant generation*, excluding the time to read (write) the files from (to) the server/disk. The time to generate the mutants using LittleDarwin or DiMuTesTas for all worker configurations should be the same, as the same amount of work should be done. While this is the case for the project from HealthConnect, we see a decrease in execution time for the project from Intris. The shorter execution time using fewer workers is due to a memory limitation when needing to process multiple large files. With more workers, each worker needs to process fewer of these files, allowing for more memory to be used for each file.

The project from Intris has fewer mutants but a higher LOC/test file than the project from HealthConnect (Table A.1). However, the mutant generation time of Intris is 3.6 times as long (see Table A.4). We assume this behaviour is caused by the complexity of generating mutants. With LittleDarwin this complexity is exponential compared to the number of lines in a file. In general, if two projects have the same LOC count, but differ in the number of files, then the one with the most files (less LOC/file) will result in a faster mutant analysis.

Mutant execution

The fourth row in Table A.4 shows the time needed to *mutant execution*, here as well excluding the time to read (write) the files from (to) the server/disk. We observe execution times which are more or less similar, which should be the case as the code responsible for this in DiMuTesTas is the exact same code as of LittleDarwin. Not surprisingly, the mutant execution time comprises the largest chunk of the whole mutation testing time, hence that is where optimisations should focus on.

RabbitMQ delay

The fifth row in Table A.4 shows the *RabbitMQ (scheduling) delay*, i.e. the time needed to gather the names of the source files, send them to the task queue plus the time needed for the workers to pull tasks from the task queue. This delay is calculated by removing all timed functions (delays, mutant generation and execution) from the local execution time of the worker. The delay incorporates the execution time of some small, untimed functions. We can see that the delay increases, but it is important to note that this is cumulative. If e.g. seven out of the eight workers are done processing, then for every second that passes before the last worker is done, seven seconds are added to this delay. If we divide the delay by the number of workers, then we see that this delay is smaller than the time it takes to process a single task. We conclude that this delay is caused by idle-time when some nodes stop processing earlier than others.

File server/disk delay

The sixth row in Table A.4 shows the *file server/disk delay*, i.e. the time needed to copy data files back and forth between the different nodes. Although this delay is relatively small, it grows linearly with the number of workers, hence is a point for concern. This delay could be minimised by sending deltas of the files over the network instead of sending the complete file.

Based on detailed measurements concerning delays in the analysis we point out directions for further optimisation. In particular, the use of *multicast* should ensure that the *set-up delay* —the current bottleneck– would take constant time, regardless of the number of nodes in the system. In the same vein, we can minimise the *file server/disk delay* by only sending the deltas of the files.

A.6 Lessons Learned

In this section we will derive the lessons learned geared towards the mutation testing community.

Nightly Builds. The mutation testing algorithm in Algorithm A.1 is inherently parallel, thus a cloud solution allows to share the computational load across a series of hardware nodes. Given sufficient hardware it is possible to off-load a full-scale mutation testing on dedicated hardware.

☑ Cloud infrastructure allows to speed up mutation testing depending on the number of hardware nodes available. This speed-up allows to perform mutation testing during the *nightly build.*

Cloud Technology. We observed that it is beneficial to run many workers on the same node to reduce the *setup delay*. However, the set-up delay is still significant and grows with the number of nodes, mainly because we copy the whole project source code to each of the available nodes. In the future, we intend to use *Multicast* the setup delay should become a constant, independent of the number of nodes in the system. In a similar vein, we will reduce the file server/disk delay by sending deltas of the files over the network.

☑ There is a lot of research and development on cloud infrastructure and the field is making rapid progress. In the near future, we may expect new features that can be exploited to make a cloud based mutation testing even faster.

Side-effects. During *mutant generation* we noticed that with more workers, each worker needs to process fewer of these files, allowing for more memory to be used for each file.

☑ Deploying mutation testing in the cloud sometimes lead to side-effects having a positive impact on the execution time.

Completely independent tasks. DiMuTesTas is designed to minimise idle times between tasks. The single task queue does not add any delay because it only needs to pass the name of the mutated file, and only the mutated file will be copied from the file server to the worker. This can only work when there are no other dependencies between tasks, in particular between the generation of the mutants (Phase A in Algorithm A.1) and the execution (Phase B in Algorithm A.1). Likewise, executing a single mutant (line 14 in Algorithm A.1) should not affect any other executions.

☑ For optimal deployment in a cloud infrastructure, tasks should remain completely independent. Mutant optimisation techniques that violate this principle will benefit less from deploying in the cloud.

Complementary Optimisation. Deploying mutation testing in the cloud reduces the total mutation testing time according to the number of hardware nodes available. Nevertheless, the *mutant generation* and *mutant execution* phases take the largest proportion of the whole analysis.

☑ There is ample room for complementary optimisation techniques that reduce the time needed to generate and execute mutants.

A.7 Threats to Validity

As with all empirical research, we identify those factors that may jeopardise 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 [81, 82]), we organise them into four categories.

Construct validity: do we measure what was intended? In essence, we wanted to know which parts of the distributed mutation testing process were causing extra delays. Therefore, we compared execution times on phases where we suspected delays could occur.

One may conceive other choices for measuring these delays. However, the suggestions for further improvement (i.e. multicast and sending of deltas of files) are likely to remain valid.

Internal validity: are there unknown factors which might affect the outcome of the analyses? As the performance of a (distributed) system can be influenced by external factors, a replication experiment could end up with different timings. For example, if the computer cannot sufficiently dissipate the generated heat by the CPU, the CPU could slow down over time. Similarly, the condition of the hard drive in the file server and the load of the network could affect the measurements. However, the results would need to differ significantly before they would invalidate the suggestions for further improvement.

External validity: to what extent is it possible to generalise the findings? We evaluated our proof-of-concept distributed mutation testing tool to industrial cases, both a small and a large one. We assume that similar measurements would apply on other projects, however the proposed solutions will need to be tailored to the projects. Projects that are small in size but have long running test suites are different from projects large in size but with very short running test suites.

Reliability: is the result dependent on the tools? We deliberately choose a mutation testing tool which clearly separates the different steps of the mutation testing process. This allowed us to measure the delay of running the mutation tests in the cloud. If the mutation testing tool contains optimisations which can be distributed without a negative impact on the performance (i.e. test prioritisation), then the results should be similar. However, as mentioned in Section A.6 For optimal deployment in a cloud infrastructure, tasks should remain completely independent.

A.8 Conclusion

In this paper, we demonstrated that cloud infrastructure allows to speed up mutation so much that it can be performed during the *nightly build*. For a small scale system (8 kLOC production code, 300 LOC test code) we could reduce the full mutation test run from 2.7 hours to 13.5 minutes. For a large project (48 kLOC production code, 5k LOC test code) we could reduce from 48 hours to 3.7 hours. As such, our proof-of-concept achieved a speed-up between 12x and 12.7x on a cloud infrastructure with 16 nodes.

Despite these improvements, there are still opportunities for further optimisation. Based on detailed measurements concerning delays in the analysis, we point out directions for further optimisation. In particular, the use of *multicast* should ensure that the *set-up delay* —the current bottleneck– would take constant time, regardless of the number of nodes in the system. In the same vein, we can minimise the *file server/disk delay* by only sending the deltas of the files.

Moreover, we also derive a few lessons learned for the mutation testing community. Most important is the principle that for optimal deployment in a cloud infrastructure, tasks should remain completely independent. Mutant optimisation techniques that violate this principle will benefit less from deploying in the cloud. Nevertheless, there is ample room for complementary optimisation techniques that reduce the time needed to generate and execute mutants.

A.9 Acknowledgments

This work is supported by (a) the ITEA TESTOMATProject (number 16032), sponsored by VINNOVA – Sweden's innovation agency; (b) Flanders Make vzw, the strategic research centre for the manufacturing industry.

Paper B

Goal-Oriented Mutation Testing with Focal Methods

Sten Vercammen
University of Antwerp
Antwerp, Belgium

Mohammad Ghafari
University of Bern
Bern, Switzerland

Serge Demeyer
University of Antwerp
Antwerp, Belgium

Markus Borg
RISE SICS AB
Lund, Sweden

Abstract

Mutation testing is the state-of-the-art technique for assessing the fault-detection capacity of a test suite. Unfortunately, mutation testing consumes enormous computing resources because it runs the whole test suite for each and every injected mutant. In this paper we explore fine-grained traceability links at method level (named *focal methods*), to reduce the execution time of mutation testing and to verify the quality of the test cases for each individual method, instead of the usually verified overall test suite quality. Validation of our approach on the open source Apache Ant project shows a speed-up of 573.5x for the mutants located in focal methods with a quality score of 80%.

Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, 2018

B.1 Introduction

Software testing is the dominant method for quality assurance and control in software engineering [10, 11], established as a disciplined approach already in the late 1970's. Originally, software testing was defined as *"executing a program with the intent of finding an error"* [12]. In the last decade, however, the objective of software testing has shifted considerably with the advent of continuous integration [13]. Many software test cases are now fully automated, and serve as quality gates to safeguard against programming faults.

Large-scale test automation is now a common practice among mature software-intensive businesses. For example, Microsoft reported that approximately 11 months of development on Windows comprised more than 30 million test case executions and Google stated that *"In an average day, TAP integrates and tests [...] more than 13K code projects, requiring 800K builds and 150 Million test runs."* [14]. By adopting high quality software testing in the continuous integration context, software companies are now releasing software much more frequently. Examples include Tesla, uploading new software in their cars once every month [15], and Amazon, pushing new updates to production every 11.6 seconds [16].

Test automation is a growing phenomenon in industry, but a fundamental question remains: How trustworthy are these automated test cases? Mutation testing is currently the state-of-the-art technique for assessing the *fault-detection capacity* of a test suite [22]. The technique systematically injects faults into the system under test and analyses how many of them are killed by the test suite. In the academic community, mutation testing is acknowledged as the most promising technique for automated assessment of the strength of a test suite [23]. One of the major impediments to industrial adoption of mutation testing is the computational costs involved: each individual mutant must be deployed and tested separately [22].

For the greatest chance of detecting a mutant, the entire test suite is executed for each and every mutant [40]. As this consumes enormous resources, several techniques to exclude test cases from the test suite for an individual mutant have been proposed. First and foremost, test prioritisation reorders the test cases to first execute the test with the highest chance to kill the mutants [45]. Second, program verification excludes test cases which cannot reach the mutant and/or which cannot infect the program state [55]. Third, (static) symbolic execution techniques identify whether a test case is capable of killing the mutant [56, 57]. This paper explores an alternative technique: fine-grained traceability links via *focal methods* [83].

By using focal methods, we can establish a traceability link at method level between production code and test code. This allows us to identify which test cases actually test which methods and vice versa. The greatest advantage of this technique is that if we know which test cases focus on testing which methods, we do not need to execute the whole test suite nor test cases that only cover a method. This allows us to drastically reduce the scope of the mutation analysis to a fraction of the entire test suite by executing only those test cases that actually test the methods of interest. This technique can also be used to quickly investigate how well a single method is tested by only executing the mutants of that method with the (few) test cases that actually test the method. We refer to this as goal-oriented mutation testing, and argue that the approach could be used to selectively target the parts of a system where mutation testing would have the largest return on investment.

To investigate the potential of focal methods in the context of mutation testing, we formulate the following research questions.

- **RQ1:** *To what extent, using focal methods, can we identify the right mutants for a test case and vice versa?*
- **RQ2:** *How much speed-up is gained by using focal methods for mutation testing?*

We validate this concept on the unit testing level using a large open source project: Apache Ant [`https://ant.apache.org`].

The rest of the paper is structured as follows. In Section B.2, we elaborate on the concept of mutation testing and general optimisations. In Section B.3, we describe the motivation for using focal methods, and how they work. In Section B.4, we explain our case study setup and discuss the results. In Section B.5, we elaborate on related work. As with any empirical research, our study is subject to threats to validity – we list the most important in Section B.6. Finally, we arrive at a conclusion in Section B.7.

B.2 Background

This section explains why we need mutation testing, why it needs to be optimised, and which types of optimisations exist.

B.2.1 Mutation Testing

Software teams need effective test cases to maximise the likelihood of exposing faults [12]. Traditionally, code coverage has been used to assess the strength of a test suite, revealing which parts of the production code are inadequately tested. Unfortunately, research has shown that code coverage might be a poor indicator of test effectiveness [17, 19]. On top of that, even a 100% MC/DC coverage (Modified Condition/Decision Coverage, the coverage criterion mandated by safety standards used for certification of safety-critical systems) does not guarantee the absence of faults [20, 21].

Mutation testing is today the state-of-the-art technique for assessing the *fault-detection capacity* of a test suite [22, 23]. By deliberately injecting faults (called *mutants*) into the production code, and counting how many of them are killed by the test suite, mutation testing has been shown to outperform traditional code coverage approaches. Case studies with safety-critical systems demonstrate that mutation testing could be effective where traditional code coverage analysis and manual inspections fail [33, 34]. Furthermore, Google reports that mutation testing both can provide insight into poorly tested parts of the system, and also reveal design problems with modules that are difficult to test, i.e. mutation testing can identify candidates for refactoring [35].

Still, mutation testing is rarely adopted in industry practice [77]. One of the reasons is that mutation testing traditionally is computationally expensive, as the code base must be compiled and tested separately for each mutant [22]. Algorithm B.2 shows the fundamental steps of mutation testing. As a prerequisite for mutation testing, referred to as the *pre*-phase, the software system needs to build without errors, and all software test cases should execute successfully. Subsequently, the two main phases are executed: (A) the mutant generation phase and (B) the mutant execution phase. In phase A, mutants are generated for all source code files. In phase B, test cases for each mutant are executed and the result (whether or not the mutant was killed) is saved. As a final step, the *post*-phase, all results are collected and the final report is created.

B.2.2 Mutation Testing Optimisations

A lot of research is devoted to optimising the mutation testing process, summarised under the vision - *do fewer*, *do smarter*, and *do faster* [42].

The *do fewer* approaches minimise the execution time by reducing the total number of mutants to execute. Such an optimisation can be implemented by generating fewer mutants on line 9

Algorithm B.2 Pseudocode Mutation Testing

```
 1: function mutationTesting(srcFolder $src$)
 2:     ▷ Pre: verify build and if all tests succeed
 3:     if initialBuildAndTests() $\neq$ True then
 4:         return
 5:
 6:     ▷ A: generate mutants
 7:     $mutants \leftarrow []$
 8:     for all srcFile $f \in src$ do
 9:         $fMutants \leftarrow$ generateMutants(srcFile $f$)
10:         $mutants \leftarrow mutants + fMutants$
11:
12:     ▷ B: execute mutants
13:     for all mutant $m \in mutants$ do
14:         $result \leftarrow$ executeMutant(mutant $m$)
15:         storeResult($result$, mutant $m$)
16:
17:     ▷ Post: process results
18:     processResults()
```

in Algorithm B.2 or by selecting a subset of all mutants on line 13. The fewer mutants that are executed, the more information will be lost. Balancing time reduction versus information loss is key. There are different ways to choose which mutants will be executed, varying in their effectiveness compared to the full set of mutants [22].

Do smarter approaches attempt to minimise the execution time by retaining state information between runs, e.g. split-stream mutation testing [47]. Another example is *test prioritisation*, which gives priority to the test cases with the highest likelihood of failure. These optimisations would be implemented on line 14 in Algorithm B.2.

Lastly, *do faster* approaches try to minimise the execution time of each individual mutant. One example is using a compiler integrated technique, where the project is compiled only once instead of for each mutant [44]. These optimisations would also be implemented on line 14 in Algorithm B.2.

Currently, optimisation techniques with large speedups sacrifice accuracy [22, 23]. Thus when evaluating mutation optimisation techniques, the trade-off between speed-up and accuracy must be quantified.

B.3 Goal-oriented Mutation Testing

Mutation testing and mutation testing optimisations have focused on detecting the *overall* quality of the test suite as fast as possible. We, however, propose a more focused approach to mutation testing, where only the test cases that actually test a method are considered.

B.3.1 Motivation

Finding the root cause of a fault is not an easy task, an entire field of study is dedicated specifically to this problem. One solution is spectrum-based fault localisation. It tries to locate the faulty component by cross referencing the test cases which detect the fault and which components are used in those test cases. One of the most recent advances uses a new metric called DDU (Density-Diversity-Uniqueness) to assess the diagnostic ability of a test suite [84]. The authors state that *"[t]he metric, tries to emulate the properties of calculating per-test coverage entropy, to ensure accurate diagnosability. Ideal diagnostic ability can be proved to exist when a suite reaches maximum entropy ... DDU focuses on three particular properties of entropy by ensuring that a) test cases are diverse; b) there are no ambiguous components; c) there is a proportional number of test cases of distinct granularity; while still ensuring tractability."* They observed a statistically significant increase in diagnostic performance of about 34% when locating faults by optimising DDU compared to branch-coverage.

In essence, this avoids the *eager test* code smell [85], i.e. testing too many methods of an object in a single test case. To increase the diagnosability of the test cases, this and other forms of *test smells* should be avoided [85]. From this we can conclude that it is best to test a method f in a test case that is specifically designed to test f and not in a test case that just so happens to call the method f in one of its routines.

If method f would be faulty, then *testF* is responsible to detect this. If f calls methods a, b, and c, then *testA*, *testB*, and *testC* are responsible for detecting faults in their respective methods. Therefore, *testF* is not required to detect a fault if the fault is inside method a, b, or c. We thus argue that given a faulty method f, it should suffice to only execute those test cases which are responsible for testing the method f. Finding which test cases are responsible to test a method can be achieved using *focal methods* [83].

B.3.2 Focal Methods

Method invocations within a test case play different roles in the test. A majority of them are ancillary to a few ones that are intended to be the actual (or focal) methods under test. More particularly, unit test cases are commonly structured in three logical parts: setup, execution, and oracle. The setup part instantiates the class under test, and includes any dependencies on other objects that the unit under test will use. This part contains initial method invocations that bring the object under test into a state required for testing. The execution part stimulates the object under test via a method invocation, i.e., the *focal method* [83, 86] in the test case. This action is then checked with a series of inspector and assert statements in the oracle part that controls the side-effects of the focal invocation to determine whether the expected outcome is obtained.

Algorithm B.3 represents a unit test case of a savings account where the intent is to test the *withdraw* method. For this, an account to test the *withdraw* method must exist. Thus, an account is created on line 3 (in Algorithm B.3). To deposit or withdraw money to/from an account, the user must first authenticate himself (line 4). To make sure that the account has money to withdraw, a deposit must be made (line 5). If the savings account has a sufficient amount of money, the *withdraw* method will withdraw the money from the account (line 7), and the remaining amount of money in the savings account should be reduced. The latter is verified using the *assert* statement on line 11.

In the example, the intent clearly is to test the *withdraw* method. This method is the focal method as it is the last method that updates the internal state of the *account* object. The expected change is then evaluated in the oracle part by observing the result of the focal method (line 10), as well with the help of the *getBalance* method which only inspects the current balance.

Algorithm B.3 Exemplary Unit Test Case for Money Withdrawal

```
1:  function testWithdraw
2:      ▷ Setup: setup environment for testing
3:      $account \leftarrow$ createAccount($account$, $auth$)
4:      $account$.authenticate($auth$)
5:      $account$.deposit(10)
6:      ▷ Execution: execute the focal method
7:      $success \leftarrow account$.withdraw(6)
8:      ▷ Oracle: verify results of the method
9:      $balance \leftarrow account$.getBalance()
10:     assertTrue($success$)
11:     assertEqual($balance$, 4)
```

Therefore, focal methods represent the core of a test scenario inside a unit test case. Their main purpose is to affect an object's state that is then checked by other inspector methods whose purpose is ancillary. A tool to detect focal methods exists, and has been recently used for extracting API usage examples from unit test cases [86].

B.3.3 Limiting the Test Scope

Under the premise that it is the responsibility of the (few) test cases that test a focal method f to catch all faults in the method f, we can assume that it suffices to limit the test scope to these selected test cases when we are looking for faults in method f. We assume even if we exclude those test cases that only happen to call a method f in one of its routines, there ought to be a simpler test case which tests the method f as a focal method that ought to also reveal that the method is faulty as that test case bears the responsibility to test the method and not the more complex test case.

Applied to mutation testing, this means that if a mutant is located in method f, we only need to execute these (few) test cases that test the focal method f.

It is possible that some test cases that are designed to test a particular method do not kill a mutant while the complete test suite can. We can define the *quality* of the individual test cases by investigating how many of the mutants that are located in a method are killed by the test cases that are designed to test the particular method. This quality score can be calculated by dividing the number of mutants killed by this technique with the total number of mutants located in the focal methods.

Summary. We propose a more straightforward approach to mutation optimisation that incorporates the diagnostic capabilities of test suites. By only executing the test cases which actually are meant to test the method f, i.e. the focal method, we hope to drastically reduce the number of test cases needed for mutants located in method f, reducing the execution time, while retaining the fault detection capabilities of the test suite.

B.4 Case Study

In this section, we explain our case setup, why and how we want to investigate our research questions, explain our gathered results, and answer our research questions accordingly.

B.4.1 Setup

To evaluate our proof-of-concept, we generated all mutants of the Apache Ant project (version 1.9.11[1]) using LittleDarwin[2] (version as of May 3 2018[3]). Project specific details[4] about Ant can be found in Table B.1. We executed the complete test suite for each mutant and stored all of the generated reports. This allowed us to inspect which of the test cases killed which mutants. While a tool exists to automatically detect the focal methods [83, 86], we will not use the tool for our proof-of-concept study, as we want our results to be independent of the tool, and to eliminate any possible errors made by the tool. As we manually needed to investigate for each test case whether the method containing the mutant is a focal method, we only examined 423 mutants. We compare our proof-of-concept against a normal mutation execution where all test cases are considered and against a mutation execution where only all the test cases of the class from which the method that contains the mutant are considered.

We point out that we encountered some intermittent faults in the test suite. Since mutation analysis needs a passing test suite to start with, we omitted the failing test cases from our analysis. The test cases in question are located in the “ant.AntClassLoaderTest” class: testCodeSource, testGetPackage, testSignedJar, and in the “ant.taskdefs.optional.XsltTest” class: testXMLWithEntitiesInNon-AsciiPath. The error in question is a java.nio.file.InvalidPathException: “Malformed input or input contains un-mappable characters” where the path includes “ãnt” instead of “ant”.

Table B.1: Details Ant Project

Commits	14,204
Contributors	98
LOC	229,019
Test Cases	1,777
Estimated amount of effort	59 years[5]
First commit	January 2000
Mutant Generated	16,354

RQ1: *To what extent, using focal methods, can we identify the right mutants for a test case and vice versa?*

[1]`https://ant.apache.org/srcdownload.cgi`
[2]`https://github.com/aliparsai/LittleDarwin`
[3]commit id: 976ae18f6535d11bf7f66e8985fa03040c419156
[4]Gathered data and data from `https://www.openhub.net/p/ant`
[5]According to the COCOMO model

Motivation. Given the fact that we assign the responsibility to detect all possible faults in a method to a few test cases, focal methods will not detect all mutants detected by the full test suite. This can have multiple causes: a test case can be incomplete (not fulfilling its responsibilities), a test case can be missing, or the focal method approach did not establish the traceability link at method level between production code and test code.

Approach. We count the number of mutants killed by the entire test suite and we count the number of mutants killed by the reduced test suite. The difference between them is the number of mutants the latter method missed and may be considered as false negatives. However, a deeper analysis is required here as they may also indicate poorly designed test cases suffering from the *eager test* code smell [85]. Whether they are killed is up to the quality of the test cases in question as discussed in Section B.3.3. The main goal is to assess for how many mutants we can find test cases with focal methods that contain these mutants.

RQ2: *How much speed-up is gained by using focal methods for mutation testing?*

Motivation. If a mutant is detectable by a test case, then ideally that should be the only test case to be executed. If the test suite cannot detect the mutant then it is pointless to run any tests for it. Reducing the scope of the test suite ensures that when the mutant is not detectable, fewer resources are wasted on the mutant. Furthermore, the fewer test cases are considered, the earlier the test case that detects the mutant is executed and the faster the mutation testing tool becomes.

Approach. For any mutant, we count the number of test cases for which the method that contains the mutant occurs as a focal method. Naturally, for the full test suite based mutation execution, all test cases are considered, and for the class based mutation execution, all test cases from that class are considered. We also count the amount of time each test case takes until the first test case that detects the mutant, this for all three mutation testing techniques.

B.4.2 Results

Table B.2: Results Focal Method Mutation Testing

Ant Class	Technique	Focal Mutants Detected	False Negatives	AVG Tests Considered	Run Time	Speed-up
Class-Loader	Full Test Suite	11 / 20	0	1,777.0	3,113.238 s	N.A.
	Class Based	10 / 20	1	9.0	9.690 s	321.3x
	Focal Methods	9 / 20	2	1.8	3.082 s	1,010.1x
Default-Logger	Full Test Suite	4 / 4	0	1,777.0	6.287 s	N.A.
	Class Based	4 / 4	0	1.0	0.010 s	628.7x
	Focal Methods	4 / 4	0	1.0	0.010 s	628.7x
Directory-Scanner	Full Test Suite	15 / 17	0	1,777.0	785.979 s	N.A.
	Class Based	11 / 17	4	29.0	7.221 s	108.8x
	Focal Methods	11 / 17	4	24.7	4.486 s	175.2x
Intro-spection-Helper	Full Test Suite	14 / 14	0	1,777.0	459.627 s	N.A.
	Class Based	11 / 14	3	14.0	0.433 s	1,061.5x
	Focal Methods	11 / 14	3	1.1	0.034 s	13,518.4x
Total	**Full Test Suite**	**44 / 55**	**0**	**1,777**	**4,365.131 s**	**N.A.**
	Class Based	**36 / 55**	**8**	**15.9**	**17.354 s**	**251.5x**
	Focal Methods	**35 / 55**	**9**	**8.6**	**7.612 s**	**573.5x**

Table B.2 indicates how many mutants we found to be located in methods which we detected as focal methods. It also indicates how many of the mutants were detected by the test suite. The *false negatives* indicate mutants that are not killed due to the limited number of test cases considered by the used techniques, but that would have been killed by the full test suite. *AVG tests considered* indicates how many test cases the technique can execute (on average) to detect the mutants. *Run time* indicates the time needed to execute all mutants (either until the mutant is detected or all considered test cases are executed) and finally, *speed-up* indicates how much faster the technique is compared to running the complete test suite.

The *full test suite* technique indicates a normal mutation testing technique where the entire test suite is considered to detect the mutant. The *class based* technique indicates a mutation testing technique where only the tests of the class in which the mutant is located is considered to detect the mutant. We included this to give our *focal method* approach some perspective. Table B.2 contains detailed information on the gathered test classes and a global overview.

RQ1: *To what extent, using focal methods, can we identify the right mutants for a test case and vice versa?*

In total, we examined the first 423 mutants out of 16,354. For 55 of them (13%), we detected test cases for which the methods containing these mutants are identified as focal methods (see Table B.2). Currently, the 368 mutants which are not found in test cases with focal methods are due to missing tests. This low recall rate is due to the way private methods are tested. In our anecdotal experience, most developers test private methods indirectly by calling public methods which act upon them. However, the current definition of focal methods is able to identify private methods as focal methods only when such methods are executed more directly, e.g. using Java reflections[6]. In future, we plan to extend the requirements of focal methods to allow indirect private methods to become focal methods.

Of the 55 mutants located in focal methods, we see that the *focal method* based approach detects fewer mutants than the *class based* approach, and thus has more false negatives. The *class based* and *focal method* based technique detect respectively 82% and 80% of the mutants the full mutation testing technique detects. This is due to incomplete test cases. While some of these mutants are detected by the test suite, they are not detected by the test cases that have the responsibility to detect them. Therefore, the quality of the individual test cases in which the detected focal methods are located can be said to be 80%.

The focal method approach benefits the most when there is an elaborate test suite where all methods are individually tested. Test suites which test at a higher level than method level, i.e. procedures and integration tests will benefit less from this technique. The extend of this impact, and the possibility to adapt focal methods to cope with procedure tests will need to be investigated in a future work.

RQ2: *How much speed-up is gained by using focal methods for mutation testing?*

For the mutants located in focal methods, we see that the average amount of test cases considered for the mutants is drastically reduced, both for the *class* based as the *focal methods* based approach: 0.9% and 0.5%, respectively. On average, the *focal methods* based approach considers half of the number of tests the *class* based approach considers. However, this highly depends on the number of tests for the class. The more tests per class, the faster the *focal methods* based approach can be compared to the *class* based approach (see AntClassLoader and IntrospectionHelper in Table B.2).

[6]`https://docs.oracle.com/javase/tutorial/reflect/index.html`

We see that the total run time of these mutants is drastically reduced as well, both for the *class based* as the *focal methods* based approach: respectively 0.4% and 0.2%.

There are two ways to look at these speed-ups, on the one hand, we can say that this currently only impacts 13% of the investigated mutants. Therefore, the speed-up considering all mutants is currently only 1.15x. On the other hand, we can say that for the methods in which 87% of the mutants are located, the test cases that have the responsibility to detect all faults in them are missing.

The current definition of focal method requires test cases to directly call a method for it to be considered as a possible focal method. In our anecdotal experience, most developers test private methods indirectly by calling public methods which act upon them. Therefore, we plan to extend the definition of focal methods to allow indirect private methods to become focal methods as well. This will increase the number of detected mutants and provides a drastic speed-up.

⇒ Focal methods can be used as a viable alternative to drastically reduce the test scope and run time of mutation testing, but improvements are needed to better cope with private methods.

B.5 Related Work

The RIPR model (Reachability, Infection, Propagation, Revealability) [87] states that in order to reveal a fault, a test case must a) reach the faulty statement (Reachability), b) cause the program state to become faulty (Infection), c) propagate the fault to the program output (Propagation) and d) cause a failure, i.e. the faulty state is asserted by the test case to its intended state (Revealability).

Three different kinds of mutation testing can be linked to this model: *weak*, *firm*, and *strong* mutation testing. For *weak* mutation testing, only the first two conditions of the RIPR model need to be satisfied. This means that a mutant is considered detected from the moment the program state of the original program and the mutated program differ. With *firm* mutation testing, an extension of *weak* mutation testing, the user can decide which component of the program state should differ from the original for a mutant to be considered as detected. Lastly, for *strong* mutation testing, all conditions of the RIPR model need to be satisfied. This means that a mutant must influence the observable output of the program (the test oracle), and not just the program state or a component in it.

Empirical evidence [40] has shown that *strong* mutation testing is more powerful than weak and firm mutation testing. In essence, to detect a mutant, at least one test case of the entire test suite should fail. For faster mutation testing, one can choose to stop executing test cases as soon as a test case kills the mutant. Ideally, we only execute those test cases for which all conditions of the RIPR model are satisfied. Test cases that do not satisfy these conditions can be excluded for performance optimisations. In the next section, we list existing techniques that consider the RIPR model to optimise for strong mutation testing.

B.5.1 Existing Techniques

Since mutation testing is such a computationally expensive technique, many researchers have sought mitigation strategies [22]. Many approaches originate in work on test suite minimisation, a set-cover problem that has been shown to be NP-complete – but several approximation solutions have been proposed [52]. As an example, Jeffrey and Gupta presented a test suite reduction technique with selective redundancy, a slightly more conservative approach (i.e. less reduction) that retains more of the fault detection effectiveness of the original test suite compared to previous work. Nevertheless, test suite reduction always requires a trade-off between execution time and fault detection effectiveness [53].

Several regression test selection methods have been proposed to speed up mutation testing, aiming at restricting test case execution to those that target the code changes. Regression test selection methods are either dynamic (i.e. using execution information) or static (i.e. based entirely on source code analysis). Chen and Zhang performed an extensive empirical evaluation of several state-of-the-art regression test selection methods for mutation testing on 20 GitHub projects [54], and conclude that the techniques are generally feasible on a file level but not for finer-grained analysis. Also, the methods studied are intended for evolving systems and not for a single version of source code.

Zhang *et al.* focused on speedup of mutation testing that works for a single source code version [45]. They developed FaMT (Faster Mutation Testing) as an approach to prioritise and reduce the number of test cases to execute for each mutant. Inspired by research on regression test prioritisation, FaMT reorders the test cases in a way to kill the mutant earlier. Subsequently, inspired by previous work on test suite reduction, FaMT runs only the subset of test cases with a high likelihood to kill the mutant. Thus, FaMT might under-approximate the mutation score – some of the skipped test cases might indeed have killed the mutant if they were executed.

There are also other approaches to exclude test cases from a test suite targeting a specific mutant. Bardin *et al.* proposed program verification to exclude test cases that cannot reach the mutant and/or that cannot infect the program state [55]. Other authors have explored using (static) symbolic execution techniques to identify whether a test case can detect mutants [56, 57]. An example of a tool implementing this approach is PIT [58], that executes only those test cases that have a chance to kill the mutant, i.e. the test cases that execute the faulty statement (thus fulfilling Reachability).

B.5.2 Comparison

As seen, there already exist techniques to exclude test cases from a test suite to speed up mutation testing. The ultimate goal for these techniques, however, is to detect all the mutants that the entire test suite can detect with as few test cases as possible. In Section B.3.1 we have argued the need for mutation testing to focus on detecting the quality of each method and its corresponding test cases instead of the overall quality of the test suite. Focal methods meet our needs, as their goal is to test the methods with the test cases that are specifically designed to test them, thus validating the quality of each test case individually.

Furthermore, focal methods deviate from the RIPR model as we will often exclude test cases that reach the faulty statement, test cases that are infected by the faulty statement, test cases that propagate the faulty statement to the program output and/or test cases that would cause a failure due to the faulty statement, but that do not have the responsibility of detecting the injected fault.

B.6 Threats to Validity

As with all empirical research, we identify those factors that may jeopardise 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 [81, 82]), we organise them into four categories.

Construct validity: did we measure what was intended? In essence, we wanted to know whether focal methods could be used to reduce the test scope for mutation testing to drastically speed up mutation testing. Therefore, we investigated a part of the Ant project and verified that for the mutants located in methods we detected as focal methods in some test cases, there was an average speedup of 573.5x. However, we only found test cases with focal methods for 13% (55/423) of the investigated mutants. This low recall rate is due to the fact that this leaves out most private methods, as developers mostly test these indirectly by calling public methods. In our next version, we will extend the requirements of focal methods to include indirect private methods. This will strongly influence our current results. Using focal methods, only 35 mutants out of the 44 detectable mutants were detected. However, the 9 mutants that are detected using full mutation testing are detected in places that do not have the responsibility to detect them. These mutants are detected because they eventually altered the program state so that the fault showed up in another place. While the test cases in which these faults showed up do not necessarily have a test code smell like the *eager tests* code smell, they do test methods that are not the intent of the test case. New, dedicated test cases that have the responsibility to detect them should be written or existing ones should be expanded. It therefore does not matter that focal methods do not detect the same number of mutants as the full mutation testing technique does, as this highlights the quality of the underlying test suite.

Internal validity: were there unknown factors that might have affected the outcome of the analyses? As we omitted the results from the test cases with intermittent faults, it is possible that we falsely identified some mutants as undetected while the omitted test could have detected it. Furthermore, as we manually analysed if the method where the mutant is located occurs as a focal method in some test cases, these results are subject to human error. We, however, believe that our obtained results show the viability of focal methods to reduce the test scope for mutation testing and drastically speed up mutation testing.

External validity: to what extent is it possible to generalise the findings? The speedup of this approach comes from validating only a few test cases for each mutant (for which focal methods are found). The larger the software project and the more test cases the project has, the more beneficial this technique becomes, as the use of focal methods will always limit the considered amount of test cases to a handful. However, when the test suite has a lot of test code smells [85], this might negatively impact the speedup of this approach. E.g. with the *eager test* code smell, many methods are tested in a single test case. With this code smell, some methods are only tested indirectly, preventing them from becoming focal methods and thus preventing them to leverage the speedup of our proposed technique. Previous work that mines API usage examples from unit test cases [86], alleviated this issue by identifying focal methods within each sub-scenario of a unit test case.

In general, this technique performs better the less test code smells the test suite has. When a project has a lot of test code smells, it might be better to first focus on removing them for better maintainability and diagnosability of the test suite [85].

Reliability: is the result dependent on the tools? We explicitly chose to gather these results manually instead of using the tool to detect the focal methods [83, 86] to eliminate any possible inaccuracies of the tool. This allowed us to investigate the feasibility of this approach without relying on the tool to make the results reliable. The only downside is that by doing so, we limited the amount of investigated mutants, increasing the odds of skewed results. However, we believe that the results indicate that the use of focal methods is a viable approach for speeding up mutation testing.

B.7 Conclusion

In this paper, we argued the need for mutation testing to focus on detecting the quality of each method and its corresponding test cases instead of the overall quality of the test suite. For this, a traceability link at method level between production code and test code must be established, which we achieved by using focal methods. This allows us to identify which test cases actually test which methods and vice versa.

We argued that for better diagnosability and maintainability of the test suite, it is best to test a method f in a test case that is specifically designed to test f and not in a test case that just so happens to call the method f in one of its routines. Using focal methods, we can focus on those test cases that are designed to test specific methods and ensure their quality. This will increase the quality of each method and its corresponding test cases and not just the overall ability of the test suite to detect defects.

We demonstrated that focal methods can be used to drastically speed up mutation testing. In our limited testing of the Ant project, we observed an average speedup of 573.5x for the mutants, located in methods that we detected as focal methods in some test cases, without sacrificing accuracy. In our experiments, we noted that the test cases with the focal methods killed 80% of the mutants killed by the entire test suite. The remaining 20% could be detected by test cases that are not intended to test the method in which these faults where injected. New, and dedicated test cases that have the responsibility to detect them should be written or existing ones should be expanded. It therefore does not matter that focal methods do not detect the same number of mutants as the full mutation testing technique, as this highlights the quality of the underlying test suite – indeed the purpose of mutation testing.

Despite these improvements, there are still opportunities for further optimisation. Currently, we did not find test cases with focal methods for 368 mutants out of the 423 investigated mutants. This low recall rate is due to the fact that the technique leaves out most private methods, as developers mostly test these indirectly by calling public methods. We plan to adapt the notion of focal methods such that it incorporates indirect private method calls as well. Then, a larger percentage of the executed mutants will be drastically sped up.

Acknowledgments

This work is supported by (a) the ITEA TESTOMATProject (number 16032), sponsored by VINNOVA — Sweden's innovation agency; (b) Flanders Make vzw, the strategic research centre for the manufacturing industry.

We also gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018).

Paper C

Focal Methods for C/C++ via LLVM: Steps Towards Faster Mutation Testing

Sten Vercammen
University of Antwerp
Antwerp, Belgium

Serge Demeyer
University of Antwerp, Belgium
Flanders Make, Belgium

Lars Van Roy
University of Antwerp
Antwerp, Belgium

Abstract

Mutation testing is the state-of-the-art technique for assessing the fault detection capacity of a test suite. Unfortunately, it is seldom applied in practice because it is computationally expensive. In this paper we explore the use of fine-grained traceability links at the method level (named *focal methods*), to drastically reduce the execution time of mutation testing, by only executing the tests relevant to each mutant. In previous work for Java programs we achieve drastic speedups, in the range of 530x and more. In this paper, we lay the foundation for identifying such focal methods under test in C/C++ programs by relying on the LLVM compiler infrastructure. A preliminary investigation on a 3.5 kLOC C++ project illustrates that we can correctly identify the focal method under test for 47 out of 61 tests,

BElgian-NEtherlands software eVOLution symposium, 2021

C.1 Introduction

Software testing is the dominant method for quality assurance and control in software engineering [10, 11], established as a disciplined approach already in the late 1970's. Originally, software testing was defined as *"executing a program with the intent of finding an error"* [12]. In the last decade, however, the objective of software testing has shifted considerably with the advent of continuous integration [13]. Many software test cases are now fully automated, and serve as quality gates to safeguard against programming faults.

Test automation is a growing phenomenon in industry, but a fundamental question remains: How trustworthy are these automated test cases? Mutation testing is currently the state-of-the-art technique for assessing the *fault-detection capacity* of a test suite [22]. The technique systematically injects faults into the system under test and analyses how many of them are killed by the test suite. In the academic community, mutation testing is acknowledged as the most promising technique for automated assessment of the strength of a test suite [23]. One of the major impediments to industrial adoption of mutation testing is the computational costs involved: each individual mutant must be deployed and tested separately [22].

For the greatest chance of detecting a mutant, the entire test suite is executed for each and every mutant [40]. As this consumes enormous resources, several techniques to exclude test cases from the test suite for an individual mutant have been proposed. First and foremost, test prioritisation reorders the test cases to first execute the test with the highest chance to kill the mutants [45]. Second, program verification excludes test cases which cannot reach the mutant and/or which cannot infect the program state [55]. Third, (static) symbolic execution techniques identify whether a test case is capable of killing the mutant [56, 57]. This paper explores an alternative technique: fine-grained traceability links via *focal methods* [83].

By using focal methods, we can establish a traceability link at method level between production code and test code. This allows us to identify which test cases actually test which methods and vice versa, hence drastically reduce the scope of the mutation analysis to a fraction of the entire test suite. We refer to this as goal-oriented mutation testing, and argue that the approach could be used to selectively target the parts of a system where mutation testing would have the largest return on investment.

In previous work, we estimated on an open source project (Apache Ant) that such goal-oriented mutation testing allows for drastic speedups, in the range of 530x and more [2]. In this paper we lay the foundation for identifying such focal methods under test in C/C++ programs by relying on the LLVM intermediate code as emitted by the CLANG compiler infrastructure.

C.2 Focal Methods under Test

Method invocations within a test case play different roles in the test. A majority of them are ancillary to a few ones that are intended to be the actual (or focal) methods under test. More particularly, unit test cases are commonly structured in three logical parts: setup, execution, and oracle. The setup part instantiates the class under test, and includes any dependencies on other objects that the unit under test will use. This part contains initial method invocations that bring the object under test into a state required for testing. The execution part stimulates the object under test via a method invocation, i.e., the *focal method* of the test case [83, 86]. This action is then checked with a series of inspector and assert statements in the oracle part that controls the side-effects of the focal invocation to determine whether the expected outcome is obtained.

Algorithm C.4 represents a unit test case of a savings account where the intent is to test the *withdraw* method. For this, an account to test the *withdraw* method must exist. Thus, an account is created on line 3 (in Algorithm C.4). To deposit or withdraw money to/from an account, the user must first authenticate himself (line 4). To make sure that the account has money to withdraw, a deposit must be made (line 5). If the savings account has a sufficient amount of money, the *withdraw* method will withdraw the money from the account (line 7), and the remaining amount of money in the savings account should be reduced. The latter is verified using the *assert* statement on line 11.

In the example, the intent clearly is to test the *withdraw* method. This method is the focal method as it is the last method that updates the internal state of the *account* object. The expected change is then evaluated in the oracle part by observing the result of the focal method (line 10), as well with the help of the *getBalance* method which only inspects the current balance.

Algorithm C.4 Exemplary Unit Test Case for Money Withdrawal

```
 1: function testWithdraw
 2:     ▷ Setup: setup environment for testing
 3:     account ← createAccount(account, auth)
 4:     account.authenticate(auth)
 5:     account.deposit(10)
 6:     ▷ Execution: execute the focal method
 7:     success ← account.withdraw(6)
 8:     ▷ Oracle: verify results of the method
 9:     balance ← account.getBalance()
10:     assertTrue(success)
11:     assertEqual(balance, 4)
```

Therefore, focal methods represent the core of a test scenario inside a unit test case. Their main purpose is to affect an object's state that is then checked by other inspector methods whose purpose is ancillary.

C.2.1 Limiting Test Scope for Mutation Testing

To adopt the focal method under test heuristic for mutation testing, we assume that a given test method does not suffer from the *eager test* code smell [85]. This is good practice anyway as it increases the diagnosability of the test cases. When this assumption holds, a test method *testF* is specifically designed to test a method f and not a series of other methods (a, b, and c). If method f would be faulty, then *testF* is responsible to detect this. If f calls methods a, b, and c, then *testA*, *testB*, and *testC* are responsible for detecting faults in their respective methods. Therefore, *testF* is not required to detect a fault if the fault is inside method a, b, or c. We thus argue that given a faulty method f, it should suffice to only execute those test cases which are responsible for testing the method f.

Under the premise that it is the responsibility of the (few) test cases that test a focal method f to catch all faults in the method f, we can assume that it suffices to limit the test scope to these selected test cases when we are looking for faults in method f. We assume even if we exclude those test cases that only happen to call a method f in one of its routines, there ought to be a simpler test case which tests the method f as a focal method that ought to also reveal that the method is faulty as that test case bears the responsibility to test the method and not the more complex test case.

Applied to mutation testing, this means that if a mutant is located in method f, we only need to execute these (few) test cases that test the focal method under test f.

Summary. We propose a more straightforward approach to mutation optimisation that incorporates the diagnostic capabilities of test suites. By only executing the test cases which actually are meant to test the method f, i.e. the focal method, we hope to drastically reduce the number of test cases needed for mutants located in method f, reducing the execution time, while retaining the fault detection capabilities of the test suite.

C.3 The LLVM Compiler Infrastructure

The original tool for identifying the focal method under test was developed for Java programs [83, 86]. No such tool exists for C/C++ hence we set out to investigate whether we can adopt the LLVM Compiler Infrastructure to expose the test-to-method relationship. LLVM is a set of compiler and toolchain technologies which is generally used to provide an intermediary step in a complete compiler environment. The idea is that any high level language can be converted to an intermediary language. This intermediary language will then be further optimised using an aggressive multi-stage optimisation provided within LLVM to then be converted to machine-dependent assembly language code [88].

Specifically, for this research we focus on the intermediary language called LLVM IR, where IR stands for Intermediary Representation. This is a low-level programming language, similar to assembly, that is easily understandable and readable. Most importantly for our analysis, the LLVM IR has explicit instructions marking the reading and writing a given variable. This implies that it is easy to distinguish getter functions (which only read the variable, hence are seldom focal methods under test) from mutator functions (which write to a given variable, hence are often good candidates for serving as a focal method under test). Secondly, LLVM IR disambiguate higher level programming constructs like function overloading. Indeed, many high level programming languages have the concept of function overloading, where the same function name can have different signatures for the same function name. It is not always apparent which overload of a given function will be used and it is therefore very hard to properly disambiguate what will happen. This in contrast to LLVM IR where every function name is unique, which is a clear benefit when establishing fine-grained test-to-method relationship.

LLVM obviously also comes with some disadvantages. First of all, the LLVM files can only be linked with other LLVM files, which on its own implies that we will need the source of every library used within the project. Secondly, LLVM IR suffers from a slight loss in context. Some high-level code constructs are not present in LLVM IR code as these are not required for the optimisation for which LLVM IR is intended. One example –importantly for our analysis– is the loss of `public` or `private` distinction.

C.4 Approach

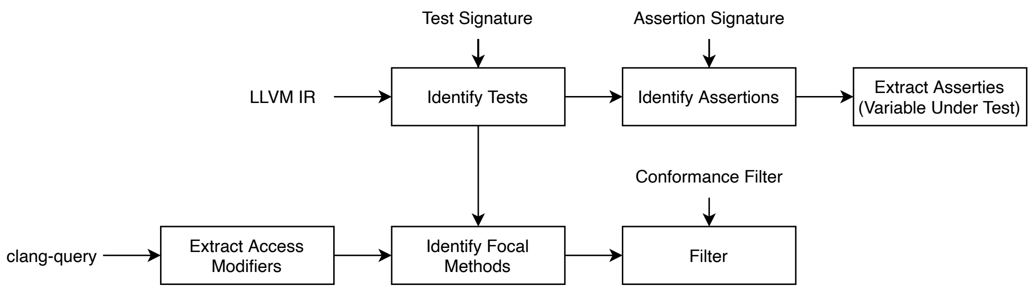


Figure C.1: Schematic Representation to Identify Focal Methods

The following section explains the approach used in the process of identifying the focal methods. The approach starts from an LLVM IR representation of the project, and results in a mapping of test methods to their corresponding focal methods under test. This approach is represented within Figure C.1.

C.4.1 Extracting Access Modifiers

As LLVM IR has no information about the access modifiers, we need to extract it beforehand. For this, we use the "clang-query" tool. For each file we store all its methods together with its acces modifier. We use this map to identify whether a method is `public` or `private` and how we need to act on it.

C.4.2 Identifying Tests

There are three major types of methods we consider for this analysis, test, assertion and source methods. A test method is distinguished from a source method by the naming convention used by the testing framework, which can simply be passed as an argument by the user. This is a necessary language dependent link as there is no fail proof way to distinguish test functions from tested functions.

Once the test methods are distinguished from all other methods, we enumerate all test methods. For each method we extract all relevant statements, in particular including all function invocations (where overloaded functions are disambiguated) and all instructions related to memory modifications.

C.4.3 Identify Assertions

To differentiate between assertions and source methods, we require a secondary input from the user, being the naming convention for assertions. These are normally implied by the testing framework used. This is usually a common prefix given to all assertion functions, eg. assert <assertion type> or use the class in which all assertions are defined, such as AssertionResult for the GTest framework.

C.4.4 Extract Asserties (Variable Under Test)

Before we can identify the focal methods for each test, we need to identify the variable under test (VUT). These are the values that are being verified in te assert statements, which we can extract these from the LLVM IR. Often these assert statements compare the VUT against a constant. Then it is easy to know which on of the two is our actual VUT. Other times, it is less obvious, as it evaluates against a variable from another class, In the latter case, we simply track both variables.

C.4.5 Identify Focal Methods

We identify the focal method by tracking the VUT throughout the invoked methods. During this process, any modification of the memory of the tracked variable will be considered a *mutator* function, hence a candidate focal method under test. It is now that the differentiation between public and private methods becomes important. We know that all candidate focal methods within the scope of the test suite are public. If one of these methods directly changes the VUT, then we label that method as a mutator. If however said method does not directly changes the VUT, but only invokes one or more public methods, then we do not analyse these public methods, as each public method should be tested by its own test according to the “no eager tests” assumption. No mutator is identified in this scenario. If however the method invokes one or more private methods, then we need to analyse these private method, as it can become the focal method, as in principle we cannot call private methods directly. We inspect all private methods within private methods, but not public methods within private methods, as each public method should be tested by its own test.

For our analysis to work with libraries for which the source code is not given, we define a third class of invocations: *uncertain*. Method invocations labelled uncertain indicate that the definition of the method is not known due to the absence of the actual implementation. Our approach resolves this by marking the function as being a potential focal method, but not the only focal method. The implication for a mutation analysis is that we will have slightly more tests linked to a mutant than when we would know the accessor modifier of the method.

C.4.6 Filter

Finally, an optional focal method conformance filter can be used. Without a means for filtering, functions defined within language specific libraries would be considered as being potential focal methods, which greatly reduces the effectiveness of the tool. For example, in C++, an assignment of string would be replaced by an assignment function defined within the standard C++ library. This function will never be the intended function under test, but considering a case where strings are compared in an assertion, the last function that will be used to assign said string to a variable will be this string assignment function defined within the std namespace. To prevent this function from being marked as the focal method, analysis of C++ code would benefit greatly from having the std namespace filtered from the set of focal methods. In our analysis, we would then still consider std library functions as functions leading to a mutation, but not as the focal methods themselves.

C.5 Proof-of-Concept and Findings

We did a preliminary investigation with our proof-of-concept tool on an extended version[1] of the Stride project (version 1.18.06[2]). The details of which are in Table C.1.

Table C.1: Details Stride Project

Lines of code	3,776,986
Number of functions	54,811
Number of test functions	222

This project was chosen because it is written in a layered manner, where only the most outer layer is accessible. The classes located on the most outer layer will be responsible for all classes directly below them, the classes below the outer classes will be responsible for the classes below those and so on. This structure is interesting, as it implies that test cases testing lower layers must traverse several other classes before arriving at the actual mutation which we want to test, making it an ideal candidate for our approach, as the method under test of those tests will be a private function.

A manual inspection of the test code revealed that of the 222 tests, 77 serve as utility tests, 59 as i/o tests, 61 as population and generation tests and 25 as scenario tests. We list these in Table C.2. Note that considering the size of the project, the test size may seem very limited, however, many of the functionalities part of the project are not testable on their own, due to the design choices made.

For now, we focused on the population and generation tests, as these adhere to the “no eager tests” assumption. More importantly, these 61 tests are tests in which the method under test should be a private function in a lower layer. This selection allows us to more precisely determine the percentage of focal methods we can correctly identify in case a private method is the method under test. If we would not make this selection, the majority of the tests would be tests where the focal method is public. These tests are less interesting for our approach as we already demonstrated that the approach is capable of detecting the mutations in such instances.

Furthermore, a part of the utility tests were tests in which an inspector method was tested. Such tests directly tests getters and might not even contain any mutations at all. Our approach cannot currently identify such tests correctly. We also disregarded scenario tests, as our approach is specifically intended to identify the method under test in unit tests, as scenario tests often test a list of methods, rather than one single method. We will have to investigate later whether our approach can be used for scenario tests.

Table C.2: Test classification Stride project

Utility tests	77
Input/Output tests	59
Population and generation tests	61
Scenario tests	25

For 47 of our 61 tests, we identified the correct focal methods. Of the remaining 14 tests, 8 were misclassified because they corresponded with “no throw assertion tests”. Such tests only confirm that no exception is thrown when the object under test is manipulated. For the remaining 6, we

[1]https://github.com/larsvanroy/stride
[2]https://github.com/broeckho/stride

identified the wrong focal method because the variable under test (VUT) where manipulated via pointers and this was not yet included in our LLVM IR Analysis.

C.5.1 Current Limitations

Library

functions will be labeled as uncertain. This can negatively impact the speedup of a mutation testing analysis using focal methods.

Inspector test

are currently not supported. We should be able to remedy this by taking into account that when no mutators are present, the last accessor method of the VUT, i.e. the inspector method should be considered as the focal method.

No throw assertion tests

are currently not supported as they do not have actual VUTs. We should however be able to detect these scenarios. We will need to investigate how or which mutants we best link to them.

Pointer support

in LLVM is something we have not yet implemented into our detection algorithm. Adding this should allow us to more accurately detect the focal methods.

C.5.2 Future Work

This proof-of-concept illustrates that it is feasible to implement the focal method under test heuristic on top of the LLVM compiler infrastructure. However, some extensions are needed to improve the accuracy of the tool. Most importantly, we need to expand the analysis to deal with assertions and pointers. Next, we need to automatically assess whether a given test case indeed satisfies the “no eager test” assumption. If we see that the assumption does not hold for a given test case, we should just revert to full scope mutation analysis. Last but not least, we need to test the tool on a larger scope of projects to see whether we can achieve similar speed-ups as what we estimated on Java projects.

Acknowledgment

This work is supported by the Flanders Make vzw, the strategic research centre for the manufacturing industry.

Paper D

Mutation Testing Optimisations using the Clang Front-end

Sten Vercammen
University of Antwerp, Belgium
Antwerp, Belgium

Serge Demeyer
University of Antwerp, Belgium
Flanders Make, Belgium

Markus Borg
RISE Research Institutes of Sweden
Lund, Sweden

Niklas Petterson
Saab Aeronautics, Saab AB
Linköping, Sweden

Görel Hedin
Lund University
Lund, Sweden

Summary

Mutation testing is the state-of-the-art technique for assessing the fault detection capacity of a test suite. Unfortunately, a full mutation analysis is often prohibitively expensive. The CppCheck project for instance, demands a build time of 5.8 minutes and a test execution time of 17 seconds on our desktop computer. An unoptimised mutation analysis, for 55,000 generated mutants took 11.8 days in total, of which 4.3 days is spent on (re)compiling the project. In this paper we present a feasibility study, investigating how a number of optimisation strategies can be implemented based on the Clang front-end. These optimisation strategies allow to eliminate the compilation and execution overhead in order to support efficient mutation testing for the C language family. We provide a proof-of-concept tool that achieves a speedup of between 2x and 30x. We make a detailed analysis of the speedup induced by the optimisations, elaborate on the lessons learned and point out avenues for further improvements.

under revision in *Software Testing, Verification and Reliability*, 2023

D.1 Introduction

DevOps (Software Development and IT Operations) is defined by Bass et al. [89] as "*a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality*" [89]. This allows for frequent releases to rapidly respond to customer needs. Tesla, for example, uploads new software in its cars once every month [15]; Amazon pushes new updates to production on average every 11.6 seconds [16]. The enabling factor for the DevOps approach is a series of automated tests which serve as quality gates, safeguarding against regression faults.

The growing reliance on automated software tests raises a fundamental question: How trustworthy are these automated tests? Today, mutation testing is acknowledged within academic circles as the most promising technique for assessing the *fault-detection capability* of a test suite [22, 23]. The technique deliberately injects faults (called mutants) into the production code and counts how many of them are caught by the test suite. The more mutants the test suite can detect, the higher its fault-detection capability is – referred to as the *mutation coverage* or *mutation score.*

Case studies with safety-critical systems demonstrate that mutation testing could be effective where traditional structural coverage analysis and code inspections have failed [33, 34]. Google on the other hand reports that mutation testing provides insight into poorly tested parts of the system, but –more importantly– also reveals design problems with components that are difficult to test, hence must be refactored [35]. In a similar vein, a blog post from a software engineer at NFluent, comments on integrating Stryker (a mutation tool for .Net programs) in their development pipeline [36]. There as well, mutation testing revealed weaknesses in the test suite, but also illustrated that refactoring allowed for simpler test cases which subsequently increased the mutation score.

Despite the apparent potential, mutation testing is difficult to adopt in industrial settings. One of the reasons is because the technique —in its basic form— requires a tremendous amount of computing power. Without optimisations, the entire code base must be compiled and tested separately for each injected mutant [22]. During one of our experiments with an industrial code base, we witnessed 48 hours of mutation testing time on a test suite comprising 272 unit tests and 5,256 lines of test code for a system under test comprising 48,873 lines of production code [1]. Hence for medium to large test suites, mutation testing without optimisations becomes prohibitively expensive.

As a consequence, the last decades has devoted a lot of research to optimise the mutation testing process [23, 52]. One stream of work focuses on *parallelisation*, either on dedicated hardware [46] or in the cloud [1]. Another stream of work focuses on *incremental approaches*, limiting the mutation analysis to what has been changed since the previous run [43]. A third stream of work (and the inspiration for what is presented in this paper) focuses on techniques based on program analysis, for example *mutant schemata* [44] and *split-stream mutation testing* [47]. The former injects all mutants simultaneously, analysing the abstract syntax tree to ensure that the mutated version actually compiles. The latter forks the test execution from the mutation point via a combination of static and dynamic program analysis.

Mutation testing shines in systems with high statement coverage because uncaught mutants reveal weaknesses in code which is supposedly covered by tests. Safety-critical systems —where safety standards dictate high statement coverage— are therefore a prime candidate for validating optimisation strategies. In safety-critical software, C and C++ dominate the technology stack [37]. Yet this is not represented in the mutation testing community: a systematic literature review on mutation testing from 2019 reports that less than 25% of the primary studies target source code from the C language family [23]. This opens up opportunities as the C language family is a mature technology with considerable tool support available. In particular, the compiler front-end Clang that operates in tandem with the LLVM compiler back-end [90].

This paper presents a feasibility study, investigating to which extent the Clang front-end and its state-of-the-art program analysis facilities allow to implement existing strategies for mutation optimisation within the C language family. We present a proof-of-concept optimisation tool, featuring a series of representative optimisation strategies. These optimisation strategies are:

1. *exclude invalid mutants*: avoid compilation overhead from mutants that would cause downstream compilation errors.
2. *exclude unreachable mutants*: avoid execution overhead from mutants that are not reached by the test suite.
3. *mutant schemata*: where all mutants get injected simultaneously and the project is only compiled once [44]. At test execution time, the appropriate mutant is selected via a boolean flag.
4. *reachable mutant schemata*: an extension of mutant schemata which reduces the test suite to only those test cases which reach the mutant.
5. *split-stream mutation testing*: where tests are executed from the mutation point itself, by forking the process, essentially avoiding redundant executions [47].

We validate the proof-of-concept tool on four open-source C++ libraries and one industrial component. These cases cover a wide diversity in size, C++ language features used, compilation times, and test execution times. Hence, they may serve as a representative benchmark to validate mutation optimisation strategies. To illustrate the potential of Clang-based implementation of mutation optimisation strategies, we report the overhead induced and the speedup achieved, both in absolute as well as relative terms. Furthermore, we derive a series of lessons learned on the benefits and impediments of implementing mutation testing optimisations using the Clang compiler front-end.

The rest of the paper is structured as follows. In Section D.2, we elaborate on the concept of mutation testing and list related work. In Section D.3, we describe the design of our proof-of-concept tool. In Section D.4, we explain how we obtained the speedups induced by the different optimisation strategies. This naturally leads to Section D.5 where we discuss the results and Section D.6 where we derive the lessons learned. As with any empirical research validating proof-of-concept tools, our study is subject to various threats to validity and limitations which are listed in Section D.7. Finally we draw conclusions in Section D.8.

D.2 Background and Related Work

In this section, we elaborate on the concept of mutation testing, the different optimisation strategies as available in the related work and discuss existing work mutation optimisation based on the Clang front-end.

D.2.1 Mutation Testing Terminology

For effective testing, software teams need strong tests which maximise the likelihood of exposing faults [12]. Traditionally, the strength of a test suite is assessed using code coverage, revealing which statements are poorly tested. However, code coverage has been shown to be a poor indicator of test effectiveness [17, 19]. Stronger coverage criteria, like full MC/DC coverage (Modified Condition/Decision Coverage, a coverage criterion often mandated by functional safety standards that target critical software systems, e.g., ISO 26262 and DO-178C) still do not guarantee the absence of faults [20, 21]. Today, mutation testing (sometimes also named *mutation analysis*, the terms are used interchangeably) is the state-of-the-art technique for assessing the *fault-detection capacity* of a test suite [22, 23].

Killed / survived mutants. Mutation testing deliberately injects faults (called mutants) into the production code and counts how many of them are caught by the test suite. A mutant caught by the test suite, i.e. at least one test case fails on the mutant, is said to be *killed.* When all tests pass, the mutant is said to *survive.*

Mutation Coverage. A strong test suite should have as few surviving mutants as possible. This is expressed in a score known as the mutation coverage: the number of mutants killed divided by the total number of mutants injected. A high mutation coverage implies that most mutants get killed; 100% is a perfect score as the tests can reveal all small deviations. Mutation coverage is sometimes referred to as mutation score.

Table D.1: Overview of commonly used mutation operators for C and C++.

Code	**Short**	**Decription**
ROR	Relational Operator Replacement	Replace a single operator with another operator. The relational operators are <,<=,>,>=,==,! =
AOR	Arithmetic Operator Replacement	Replace a single arithmetic operator with another operator. The operators are: $+, -, *, /, \%$
LCR	Logical Connector Replacement	Replace a logical connector with the inverse. The logical connectors are: ||, &&, |, &
UOI	Unary Operator Insertion	Insert a single unary operator in expressions. Example unary operators are: increment (++), decrement (−−), address (&), indirection (∗), positive (+), negative (−), . . .
SDL	Statement Deletion	Selective deletion of code, including removing a specific function call, or replacing a method body by `void`
AMC	Access Modifier Change	Changes the access level for instance variables and methods to other access levels. Access levels are `private`, `protected`, `public`
ISI	Super Keyword Insertion (or Base keyword insertion)	Inserts the `scope resolution operator ::` so that a reference to the variable or the method goes to the overridden instance variable or method
ICR	Integer Constant Replacement	Replaces every constant c with a value from the set $\{-1, 0, 1, -c, c-1, c+1\}\backslash\{c\}$

Mutation Operators. Mutation testing mutates the program under test by artificially injecting a fault based on a mutation operator. A mutation operator is a source code transformation which introduces a change into the program under test. Typical examples are replacing a conditional operator (e.g., >= into <) or an arithmetic operator (e.g., + into −). The first set of mutation operators was reported in King and Offutt [47]. Later, special purpose mutation operators have been proposed to exercise errors related to specific language constructs, such as Java null-type errors [91] or C++11/14 lambda expressions and move semantics [92]. There are more than 100 mutation operators reported in the academic literature and there is no consensus of which ones are best for a specific language and code base. Therefore mutation testing tools feature a diverse set of mutation operators which can be configured when performing the mutation analysis. Table D.1 lists commonly used mutation operators for C and C++ which we will refer to later in this paper.

Invalid Mutants. Mutation operators introduce syntactic changes, hence may cause compilation errors in the process. If we apply the arithmetic mutation operator (AOR) to e.g. “a * b”, then we get four mutants as shown in Listing D.1. However, the modulo operator (‘%’) will give an “`invalid operands to binary expression`” error, as the modulo operator is not defined for floating point data types. The mutant can thus not be compiled and is considered invalid.

Another frequently occurring invalid mutant occurs when changing the ‘+’ into a ‘−’ which works for numbers but does not make sense when applied to the C++ string concatenation operator. If the compiler cannot compile the mutant for any reason, the mutant is considered invalid and is not incorporated into the mutation coverage. Preventing the generation of invalid mutants is one way to optimise the mutation testing process.

Listing D.1: Mutation Example

```
float f(float a, float b) {
    return a * b;   // original code
}
    return a + b;   // mutant 1
    return a - b;   // mutant 2
    return a / b;   // mutant 3
    return a % b;   // mutant 4
           ~~~^~~~Invalid operands to binary expression
```

Unreachable Mutants. The Reach–Infect–Propagate–Reveal criterion (a.k.a. the RIPR model) provides a fine-grained framework to assess weaknesses in a test suite which are conveniently revealed by mutation testing [93]. It states that an effective test should first of all *Reach* the fault, then *Infect* the program state, after which it should *Propagate* as an observable difference, and eventually *Reveal* the presence of a fault. When a mutant is injected in a statement that is never executed by the test suite, it can never be killed. Therefore, one can optimise the mutation testing process by explicitly excluding unreachable mutants. To increase its effectiveness, this should be done on the test case level instead of on the entire test suite.

Equivalent Mutants. Injected mutants can be syntactically different from the original software system, but semantically identical. These mutants do not modify the meaning of the original program, and can therefore not be detected by the test suite. Such mutants are called equivalent mutants. They yield false negatives and decrease the effectiveness of the mutation analysis. Additionally, they waste developer time, as they need to be manually labelled as equivalent mutants because they show up as survived mutants, which can never be killed. Consequently, a big challenge of mutation is handling (and/or eliminating) these equivalent mutants. An overview of techniques to overcome the equivalent mutant problem has been provided by Madeyski et al [41]. One of the frequently used techniques is called *Trivial Compiler Equivalence (TCE)*: if the compiler emits the same low-level code (machine code) then the mutant is guaranteed to be equivalent.

Timed out Mutants. Some injected mutants cause the test to go into an infinite loop. To prevent such infinite loops, mutants with an excessively long execution time need to be detected and stopped. This is often done using a time out, which terminates the test after a predefined period of time. A mutant which causes such a time-out is considered killed.

D.2.2 Mutation Testing Optimisations

To explain the time-consuming nature of the mutation testing process, Algorithm D.5 shows the essential steps of a mutation analysis without any optimisations. The software system needs to build without errors and all software tests should succeed before mutation testing can even begin; this is called the *pre*-phase. Then, the two main phases are executed: (A) the generate mutants phase and (B) the execute mutants phase. In phase A, mutants are generated for all source files. In phase B, for each mutant, all tests are executed and the result —whether or not it was killed— is saved. Finally, all the results are gathered and the final report is created in the *post*-phase.

Algorithm D.5 Pseudocode Mutation Testing

```
 1: function MUTATIONTESTING(srcFolder $src$)
 2:     ▷ Pre: verify build and if all tests succeed
 3:     if INITIALBUILDANDTESTS() $\neq$ True then
 4:         return
 5:
 6:     ▷ A: generate mutants
 7:     $mutants \leftarrow []$
 8:     for all srcFile $f \in src$ do
 9:         $fMutants \leftarrow$ GENERATEMUTANTS(srcFile $f$)
10:         $mutants \leftarrow mutants + fMutants$
11:
12:     ▷ B: execute mutants
13:     for all mutant $m \in mutants$ do
14:         COMPILEMUTANT(mutant $m$)
15:         $result \leftarrow$ EXECUTEMUTANT(mutant $m$, testsuite $t$)
16:         STORERESULT($result$, mutant $m$, testsuite $t$)
17:
18:     ▷ Post: process results
19:     PROCESSRESULTS()
```

A lot of research is devoted to optimising the mutation testing process, summarised under the principle — *do fewer*, *do smarter*, and *do faster* [42].

- *Do fewer* approaches minimise the execution time by reducing the total number of mutants to execute. Such an optimisation can be implemented by generating fewer mutants on line 10 in Algorithm D.5 or by selecting a subset of all mutants on line 15. Incremental approaches[43], limiting the mutation analysis to code changed in a commit are a particularly relevant example of a "do fewer" approach. A reduced set of mutants normally incurs an information loss compared to the full set of mutants, however, the effectiveness is often acceptable [22]. Nevertheless, when the mutation testing tool provides sufficient guarantees to identify *invalid mutants* or *unreachable mutants*, excluding these is always an effective optimisation.
- *Do smarter* approaches attempt to minimise the execution time by exploiting the computer hardware (e.g. distributed architectures, vector processors, fast memory access). The `for` loops in lines 9 and 15 in Algorithm D.5) have few data dependencies, hence can be executed in parallel. Parallel execution of mutants, either on dedicated hardware [46] or in the cloud [1] is known to speed up the process by orders of magnitude. *Split-stream*

mutation testing is the de facto representative for "do smarter" optimisations [47]. By retaining state information between test runs, split-stream mutation testing avoids the redundant execution of statements up until the mutation point.

- *Do faster* approaches attempt to minimise the execution time by reducing the execution cost for each mutant (cfr. line 17 in Algorithm D.5). By design, a mutated program is almost identical to the original program which can be exploited during the compilation step. *Mutant schemata* [44] is the best know example. All mutants get injected simultaneously (guarded by a global switch variable), hence the project is compiled only once in line 16. During the mutant execution phase (in line 17) the global switch is used to select the actual mutant to execute. The execution time of a mutant can also be reduced with *test prioritisation* techniques. By rearranging the test suite, the tests with the highest likelihood of failure will be executed first, reducing the test suite run time using early-failure [45].
- *Hybrid approaches.* Different approaches are often synergistic, where a combination of techniques becomes more than the sum of the parts. Note that some approaches are orthogonal to one another hence are easy to combine. Excluding unreachable mutants, for example, can be combined with any other optimisation. Other approaches, however, may depend on each other. Mutant schemata, for instance, requires that all invalid mutants are excluded because even a single invalid mutant will immediately invalidate the whole mutated program. Measuring the speedup of a given optimisation strategy should take these synergies into account.

D.2.3 LLVM & Clang Compiler Infrastructure

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. [...] capable of supporting both static and dynamic compilation of arbitrary programming languages. [`https://LLVM.org`]

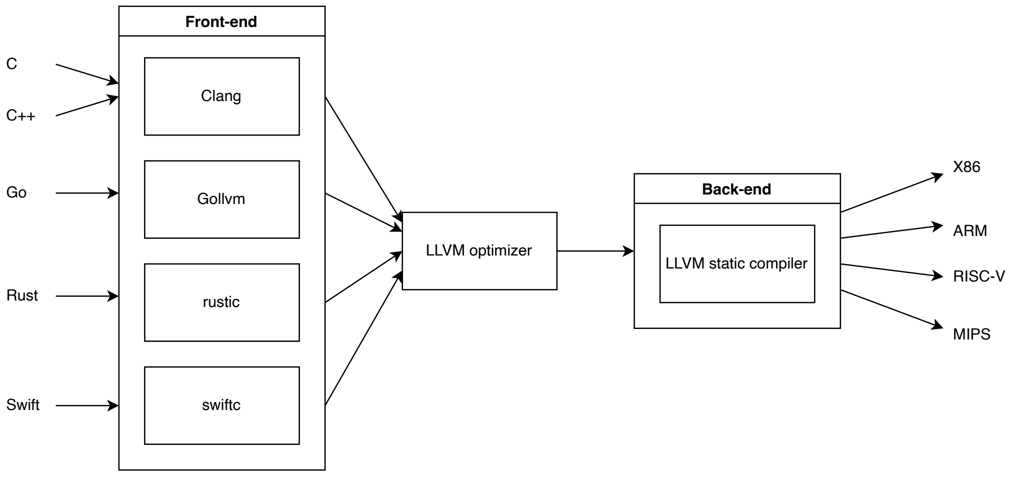


Figure D.1: LLVM Compilation

LLVM is a collection of compilation tools designed around a low-level language-independent intermediate representation, the LLVM IR. The project includes frontends that translate source code to LLVM IR, optimisers that rewrite the LLVM IR to become faster, and backends that generate machine code from the LLVM IR for different architectures. We visualised this in Figure D.1.

Clang is the most well-known frontend for LLVM, and supports languages in the C family, like C, C++, and Objective-C, among others. Internally, it represents programs as abstract syntax trees (ASTs). Clang includes a semantic analyser that does type-checking and other compile-time checks, before generating the IR. Furthermore, Clang contains a number of libraries based on the visitor pattern, allowing more analyses or transformations to be added to the frontend.

LLVM and Clang serve as the de facto standard for building static analysis tools for the C language family. It should therefore come as no surprise that several C and C++ mutation tools exist that build upon these frameworks. These tools mutate the program either at the AST level or at the LLVM IR level. Both approaches have advantages and disadvantages. Doing the mutations at the LLVM IR level has the advantage that they will work for any frontend, but the disadvantage that mutants injected in the LLVM IR are difficult or even impossible to trace back to a source representation in the original code under test, which allows for the generation of many invalid mutants.

The AST, on the other hand, is close to the source code. Mutating at this level provides good traceability, which is critically important for reporting back results to the developer in the *post*-phase (lines 18 and 22 in Algorithm D.5). Another advantage of mutating at the AST level is that the frontend semantic analyser can be used to ensure that the mutated code is compile-time correct, effectively eliminating invalid mutants.

D.2.4 Mutating on the LLVM IR and AST level

As the AST is close to the source code, the source code from Listing D.1 can be transformed into an AST without losing any information. This can be seen in Listing D.2 where even the original line and column information of the statements are retained. The AST includes all the semantic details in an easily accessible manner. This, together with the frontend semantic analyser allows for more informed mutations, allowing the detection of invalid mutations such as the modulo operator of mutant 4. Mutating the multiplication, i.e. the binary operator `'*'` on line 6, is achieved by changing the `'*'` to `'+,-,/'`. As the location information is preserved, the mutants are easily represented in the original source code. Tools mutating the AST will do so by either iterating through the entire AST or by using AST matchers which provide a list of all candidate nodes based on a set configuration e.g. BinaryOperator.

Listing D.2: AST Example

```
1  |-FunctionDecl 0x7fac9d87b710 <main.cpp:1:1, line:3:1> line:1:7 used f
       'float (float, float)'
2  | |-ParmVarDecl 0x7fac9d87b5b8 <col:9, col:15> col:15 used a 'float'
3  | |-ParmVarDecl 0x7fac9d87b638 <col:18, col:24> col:24 used b 'float'
4  | `-CompoundStmt 0x7fac9d87b8a8 <col:27, line:3:1>
5  |   `-ReturnStmt 0x7fac9d87b898 <line:2:5, col:16>
6  |     `-BinaryOperator 0x7fac9d87b878 <col:12, col:16> 'float' '*'
7  |       |-ImplicitCastExpr 0x7fac9d87b848 <col:12> 'float' <
       LValueToRValue>
8  |       | `-DeclRefExpr 0x7fac9d87b808 <col:12> 'float' lvalue ParmVar 0
       x7fac9d87b5b8 'a' 'float'
9  |       `-ImplicitCastExpr 0x7fac9d87b860 <col:16> 'float' <
       LValueToRValue>
10 |         `-DeclRefExpr 0x7fac9d87b828 <col:16> 'float' lvalue ParmVar 0
       x7fac9d87b638 'b' 'float'
```

In Listing D.3 we show our mutation example in the LLVM IR format. Line 2 to 9 represent '**return** a *b;' As this is closer to the machine code, less information from the original source code is preserved, e.g. the variable names *a* and *b* are converted to numbers *%0* and *%1*, and low-level instructions such as the allocation (alloca) of variables are introduced. Mutating the multiplication in the LLVM IR code means mutating the *fmul* statement to *fadd, fsub*, and *fdiv*. An LLVM IR mutation testing tool usually works in two steps. It first creates a list of mutation points (specific LLVM instruction and its operands). It will then create a new LLVM IR version for each mutant by applying the mutant to the mutation point.

Listing D.3: LLVM IR Example

```
define float @_Z1fff(float %0, float %1) #0 {
    %3 = alloca float, align 4
    %4 = alloca float, align 4
    store float %0, float* %3, align 4
    store float %1, float* %4, align 4
    %5 = load float, float* %3, align 4
    %6 = load float, float* %4, align 4
    %7 = fmul float %5, %6
    ret float %7
}
```

As there are many more instructions at the LLVM IR-level, they present more opportunities for mutations. However, mutations can exist in the LLVM IR code which cannot be achieved by changing the original source code [59]. As no conversion to the original code is possible, there is no traceability and no easy feedback for the developers. Additionally, not all mutation opportunities from the source code are available at the LLVM IR-level due to optimisation in the conversion to LLVM IR. In short, LLVM IR mutations provide a different set of mutants than mutating at the AST-level.

D.2.5 Existing LLVM and Clang Mutation Testing Tools

Below we discuss the most prominent mutation testing tools that are based on Clang and/or the LLVM IR. Table D.2 lists them in alphabetical order with their features and optimisations. For each of them, we briefly explain which optimisations are incorporated and refer to quantitive evidence if present.

AccMut (IR-based): AccMut [60] is an LLVM IR mutation testing tool that reduces redundant execution statements anywhere in the program by analysing the original and mutated program. It identifies the redundant statements by inspecting the (local) state of both programs. When they are identical, all following statement executions are identical and redundant until the next different statement. They have demonstrated an average speedup of 8.95x over a mutant schemata approach [60] .

CCmutator (IR-based): CCmutator [94] is an LLVM IR mutation testing tool specifically designed to mutate concurrency constructs.

Dextool mutate (AST-based): Dextool is an open-source framework created for testing and static analysis of (often safety-critical) code. The Dextool framework is used within industry, for example within Saab Aeronautics. One of the plugins in the framework is Dextool mutate. It was developed with a heavy emphasis on the reporting part of mutation testing in order to better understand the output of mutation testing and to gain more insight into the project under test.

Table D.2: Clang and LLVM IR Mutation Testing Tools

Tool Name	Mutation level	Mutation Operators	Mutation Optimisation
AccMut	LLVM IR	AOR, ROR, LCR, SDL, ...	Mutant Schemata, modulo states
CCmutator	LLVM IR	Concurrency Mutation Operator	
Dextool mutate	AST	AOR, ROR, LCR, SDL, UOI	Distribution, Mutant Schemata, Exclude unreached mutants via code coverage
Mart	LLVM IR	Operator groups	Trivial Compiler Equivalence
MuCPP	AST	Class Level Mutants	Reduced Mutants Set
Mull	LLVM IR	LLVM fragments	Limit total mutants based on call-depth
SRCIROR	AST	AOR, LCR, ROR, ICR	Trivial Compiler Equivalence, Exclude unreached mutants via code coverage

Dextool mutate works by (textually) inserting mutants into the source code one at a time after analysing the abstract syntax tree (conveniently available via Clang) for points to mutate.

Dextool mutate allows users to provide scripts and special flags in order to compile and test projects with the explicit intention to scale for more and bigger (industrial) projects. Dextool mutate supports a distributed setup as the mutants are stored in a central database. Multiple nodes can then access the database and execute a subset of all mutants. During this study, we created a proof-of-concept schemata plugin for Dextool mutate to enable mutant schemata. This plugin was requirement for our experiment in order to execute the mutant schemata approach on the Saab Case as they where already utilising Dextool. The steps in the Dextool mutate schemata plugin are identical to our standalone tool as described in this paper. The results and timings from the Saab Case were gathered using the Dextool mutate schemata plugin. As a result, the creators of Dextool created a proper implementation for mutant schemata into the tool.[1,2]

Mart (IR-based): Mart [95] is an LLVM IR mutation testing tool that currently supports 18 different operator groups (with 68 *fragments* and 816 operators). These operator groups match against the LLVM IR syntax to create the mutants. Additional operator groups can be implemented by the user to further extend its capabilities. Mart has an in-memory implementation of Trivial Compiler Equivalence to eliminate equivalent and duplicate mutants [63].

MuCPP (AST-based): MuCPP is a Clang-based mutation testing program that generates mutants by traversing the Clang AST and storing the mutants in different branches using a version control system [96]. MuCPP implements mutations at the class level. These include mutations related to inheritance, polymorphism and dynamic binding, method overloading, exception handling, object and member replacement, and more. The study is aimed at reducing the total number of mutants that need to be executed by eliminating so-called unproductive

[1]`https://github.com/joakim-brannstrom/dextool/tree/master/plugin/mutate/contributors.md`

[2]`https://github.com/joakim-brannstrom/dextool/blob/master/plugin/mutate/doc/design/notes/schemata.md`

mutants. These include equivalent mutants, invalid mutants, easy-to-kill mutants, and mutants in dead code. MuCPP demonstrates that Clang can be used for implementing mutant analysis and generation at the AST level. The study lists the speedup gained from the reduced mutation set but does not list the overhead impact of the generation and analysis of the mutants using the Clang framework. It does not implement other optimisation techniques, so there are no detailed measurements of the potential reduction in compilation and execution overhead using the Clang framework, nor the overhead the implementation of such techniques using the Clang framework might cause.

Mull (IR-based): Mull is an open-source mutation testing tool[3] which modifies fragments of the LLVM intermediate representation (LLVM IR). It only needs to recompile the modified fragments in order to execute the mutants, keeping the compilation overhead low [59]. As Mull modifies LLVM code, it is compatible with all programming languages that support compilation to LLVM IR, such as C, C++, Rust, and Swift. Mull includes a *do-fewer* optimisation where you can limit which mutants are executed to only those mutants that are within a certain call-depth starting from the test case. The study reports on the total runtime for the optimised mutation tool but does not provide details for the runtime of the individual steps nor is the tool contrasted to a traditional, unoptimised approach. While the tool certainly provides a speedup for mutation testing, the lack of detailed measurements makes it difficult to estimate where the advantages lie and where overhead might occur.

SRCIROR (AST or IR-based): SRCIROR [97] is a toolset for performing mutation testing at the AST level or at the LLVM IR level. Both variants implement the *AOR, LCR, ROR, ICR* mutation operators. When mutating the AST, SRCIROR uses the AST matchers from the Clang LibTooling library to search for candidate mutation locations in the AST. It then generates a different source file for each of the generated mutants. When mutating the LLVM IR, SRCIROR creates a list of mutation opportunities (specific LLVM instruction and one of its operands) within the generated LLVM IR code of the project. SRCIROR then creates a mutated version for each of these mutation opportunities. SRCIROR allows to filter out unreachable mutants based on code coverage metrics. It also allows to filter out some equivalent mutants using trivial compiler equivalence [63].

Summary. The current state-of-the-art demonstrates that mutant analysis for the C language family is possible on top of the LLVM and Clang compiler framework. However, the extent to which the various optimisation strategies allow reduced compilation and execution overheads is unknown. In particular, there exist no detailed measurements on two of the most advanced techniques (*mutant schemata* and *split-stream mutation testing*) for the C language family.

[3]`https://github.com/mull-project/mull`

D.3 Proof-of-Concept Tool

In this paper, we investigate to which extent the Clang front-end and its state-of-the-art program analysis facilities allow to implement existing strategies for mutation optimisation within the C language family. For this, we implement five strategies: exclude invalid mutants, exclude unreachable mutants, mutant schemata, reachable schemata, and split-stream. We published our proof-of-concept tool as an artefact on codeocean together with an accompanying description on how to use it[4].

The goal of these optimisation strategies is twofold. On the one hand they try to eliminate the compilation overhead. On the other hand, the optimisation strategies try to reduce the execution overhead. The different strategies build on each other, and successively introduce more optimisations.

The first compilation overhead comes from the invalid mutants, which cause compilation errors and can therefore not be executed. Excluding them reduces the compilation overhead. The second compilation overhead comes from individually compiling each mutant, which is computationally expensive. A mutant schemata strategy eliminates this overhead by compiling all mutants at once, drastically reducing the compilation time.

The first execution overhead comes from the unreachable mutants. These mutants are unreachable by the test suite, hence, they will always survive. If we know which mutants are unreachable, we can label them as survived without needing to execute them. The second execution overhead comes form the fact that not all test cases reach each mutant. In order for a test case to potentially kill a mutant, that test case needs to first reach and execute the target mutant. If the test case does not reach the mutant, we know that it will never be able to kill it, hence we know its result before we execute it. We created a detection algorithm that avoids this redundant execution by detecting which test case reaches which mutant, and thus only executes a subset of test cases for each mutant. The final optimisation strategy we looked into is the split-stream mutation. This strategy exploits the state-space information so that the execution of each mutant can start from the mutation point itself instead of always starting the execution at the beginning of the test suite. This essentially cuts the amount of statements that need to be executed in half.

We present a proof-of-concept optimisation tool, featuring an unoptimised approach for a baseline and the aforementioned optimisation strategies. We discuss each strategy, their differences and similarities, the potential impact on compilation and execution time and provided detailed measurements of the steps in the optimisation strategies. The general steps of the optimisation strategies are visualised in Figure D.2. We will explain each of the optimisations by its steps, starting at the bottom of the image to the top.

Generate Mutants. (Bottom boxes in Figure D.2.) The generation of the mutants is done independently of the optimisation strategies. This allows us to use the same mutants, ensuring a fair comparison of the strategies. This is represented by its inclusion in every pillar in Figure D.2.

Currently, for our proof-of-concept tool, we have implemented the Relational Operator Replacement (ROR), Arithmetic Operator Replacement (AOR) and Logical Connector Replacement (LCR) (see Table D.1). This is a representative selection of all possible mutation operators, sufficient to demonstrate the feasibility of Clang based optimisation strategies. We discuss this limitation under Section D.7.

[4]`https://codeocean.com/capsule/3514968/tree/v1`

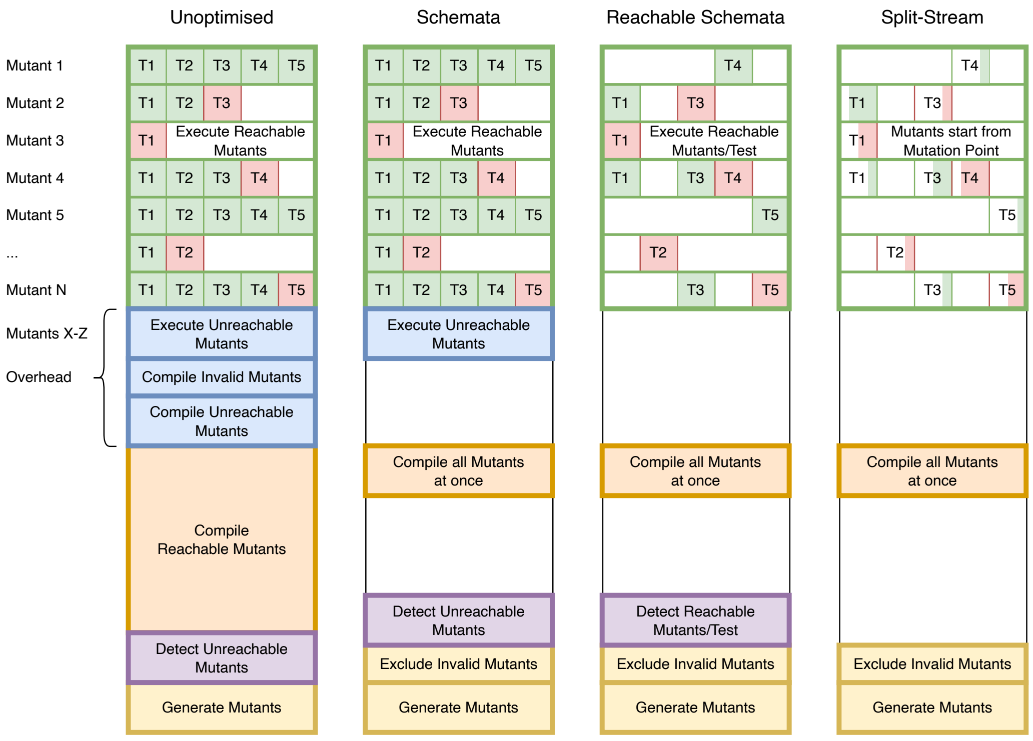

Unoptimised
Schemata
Reachable Schemata
Split-Stream
Mutant 1
Mutant 2
Mutant 3
Mutant 4
Mutant 5
...
Mutant N
Mutants X-Z
Overhead
T1
T2
T3
T4
T5
Execute Reachable Mutants
Execute Reachable Mutants/Test
Mutants start from Mutation Point
Execute Unreachable Mutants
Compile Invalid Mutants
Compile Unreachable Mutants
Compile Reachable Mutants
Compile all Mutants at once
Detect Unreachable Mutants
Detect Reachable Mutants/Test
Exclude Invalid Mutants
Generate Mutants

Figure D.2: Implementation strategies with algorithm steps (first step at the bottom). Each strategy improves on the previous one by reducing compilation and/or execution overhead.

Our proof-of-concept tool relies on the LibTooling library of Clang, and its ability to iterate through all declarations, statements, and expressions from the abstract syntax tree (AST). For each of these declarations, statements, and expressions, the implemented mutation operators will create corresponding mutants. As an example, the AOR mutation operator when encountering the multiplication '$*$' operator residing in the binary expression '$a * b$' will create the mutants where the multiplication is replaced by '$+$', '$-$', '$/$', and '$\%$' operators. To ensure that we know where to replace the original operator with the mutated one, we store the filename in which the mutant occurs, the offset to the beginning and the end of the original operator, and the mutant itself. For each executed mutant, we store whether or not the mutant was reached, how long it took to execute, and whether or not the mutant was killed. As it is possible that some mutants will cause an infinite loop, the user can set a maximum execution time for each mutant. If the test suite is not able to finish executing within this time, we stop the test execution and say that the mutant *timed out*. Mutants that are timed out can be considered as killed, as they would also time out in a continuous integration test. This information can then be extracted to form a report to inform developers where and which mutants survived the test suite.

D.3.1 Unoptimised Mutation Testing

In a traditional, unoptimised mutation testing setting each mutant is inserted, compiled and executed separately. Some mutants, however, can be located in uncovered code. Such mutants are unreachable and are unable to infect the program state, thus they can never be killed. In a realistic scenario, one would first run a code coverage technique to determine which mutants are located in uncovered code and instantly label them as survived. As the purpose of this paper is to gain more insight into the potential speedups of our optimisation techniques, we do generate and execute them in order to gain more insight into the kind of overheads these unreachable mutants introduce.

For detailed measurements, we timed the generation of the mutants, the compilation of the mutants, and the execution of the mutants. For the mutants themselves, we make a distinction between the invalid mutants (i.e. mutants that cause compilation errors) and the valid mutants. We further divide these valid mutants into the unreachable mutants (i.e. mutants that no test case reaches), and the reachable mutants.

We implemented the unoptimised mutation testing approach to obtain a baseline for the mutation analysis and to verify the correctness of our optimisation strategies.

Detect Unreachable Mutants. In order to determine which mutants are completely unreachable by the test suite, we instrument the code base with a wrapper on the mutated locations. Running the test suite with the instrumented code base then provides us with a list of mutants that are reachable by the test suite. From this, we can deduce the completely unreachable mutants.

Compile Mutants. After the generate mutants phase, each mutant needs to be individually inserted into the original code base and compiled before it can be executed. We measured the compilation time for each mutant and divided them into three categories: reachable, unreachable, and invalid mutants. This allows us to quantify the overhead caused by the unreachable and invalid mutants.

All three can be seen in the first pillar, i.e. unoptimised, in Figure D.2. It is important to note that we do not clean the build environment between the different mutants. We use consecutive builds to speed up the compilation, as the compiler then only needs to re-compile the files that are impacted by the mutant.

Execute Mutants. After generation and compilation, the valid mutants need to be executed. For each mutant, the entire test suite needs to be run. The test cases of the test suite are represented by T1 to T5 in Figure D.2. When a test case fails, represented by the red colour, this means that the mutant is killed. When none of the test cases fail, the mutant survives, represented by the green colour. To optimise the mutation testing execution, we rely on the early-stop principle, for which the test suite execution stops when the first test case kills the mutant. In the unoptimised approach, the entire test suite is run for each unreachable mutant. We represented this with mutants X–Z in Figure D.2.

Expected Performance

For the unoptimised approach, we estimate the performance via Formula D.1. The formula consists of three phases: the generate mutants, compile mutants, and execute mutants phase. We included the generation of the mutants in the formula as this step is necessary and identical for all approaches but its impact should be negligible. We do not include the detection of unreachable mutants as it is not part of an unoptimised approach. We only need it to distinguish between the unreachable and reachable mutants. The total compilation time is subject to the amount of reachable, unreachable and invalid mutants. From our measurements, we have seen that invalid mutants can take up to 10% of the total execution time. The total test suite execution time is only subject to the amount of reachable and unreachable mutants as invalid mutants cannot be executed. Note that the compilation time of consecutive builds is lower than a clean build and that the test suite execution time varies depending on where, if at all, the mutant is killed due to the early-stop mechanism.

$$\begin{array}{ll} t_{mutant\_generation} & \\ + t_{compilation} & *(reachable\_mutants + unreachable\_mutants + invalid\_mutants) \\ + t_{test\_suite\_execution} & *(reachable\_mutants + unreachable\_mutants) \end{array} \tag{D.1}$$

D.3.2 Mutant Schemata

The mutant schemata strategy compiles all mutants at once, essentially eliminating the compilation overhead [98]. Previous studies with mutant schemata have shown that this optimisation approach can provide an order of magnitude improvement on a full mutation analysis. Untch et al. reported a preliminary speedup of 4.1 [98]. In a later study, Wang et al. confirmed these findings and reported a speedup between 6.46 and 14.00 on systems written in Java [60]. However, we found no specific studies investigating the speedups of mutant schemata for the C language family, hence, in this study, we will collect detailed measurements. We timed the generation of the mutants, the compilation of all the mutants at once, and the execution of the mutants. We can then compare this to the unoptimised approach to gain insights into its speedups and potential overheads. For a fair and complete comparison, we again make the distinction between the unreachable and reachable mutants.

Exclude Invalid Mutants. As all mutants are inserted into a single compilation unit, all generated mutants need to compile. If even a single mutant causes a compilation error, the complete mutation analysis will fail. This is especially challenging for statically typed languages with many interacting features and unforgiving compilers (C, C++, . . .). This is where the Clang compiler front-end can be used for ensuring that all injected mutants will compile. As Clang has access to all the type information from the project, our program can verify, during the generate mutants phase, that a newly created mutant is compile-time correct. For this, we rely on the semantic analyser of Clang. As all the information is available while traversing the AST, the additional analysis time should be limited. If a mutant is incorrect, for example "string - string" or "float % int", the mutant is labelled as invalid, as the mutant would cause a compilation error. To ensure that we exclude only the invalid mutants, we verified that the mutants we labelled as invalid are the same mutants that actually caused compilation failures in the unoptimised baseline. As this step is tightly integrated with the generation of the mutants, we measured this together with the generation of the mutants.

Detect Unreachable Mutants. The detection of the unreachable mutants is identical to the one for the unoptimised approach.

Compile Mutants. Instead of compiling each mutant separately, the mutant schemata strategy compiles all mutants at once. This means that all mutants are inserted into the code, each mutant being guarded by a conditional statement allowing the activation of individual mutants at run-time. The result is a single code base and only one compilation is needed, drastically reducing the compilation time. We visualised this in Figure D.2 by using a smaller 'compile all mutants at once' block . A specific mutant is activated by using an external environment variable. For this, additional code, i.e. the external variable, needs to be added to each file of the project (see line 1 in Listing D.4). Listing D.4 shows the mutant schemata version of the original "$a * b$" example. We use the ternary operator, the short-handed version of an *if* statement, to allow nesting the mutants inside the condition of *if* statements.

In the example, the '%' operator gives an "`invalid operands to binary expression`" error, therefore, we cannot compile our program unless we remove this mutant. As we exclude invalid mutants during generation, these will not show up in the mutated code base.

Listing D.4: Mutant Schemata Example

```
 1  extern int MNR; // prepended external variable, allowing selection of
        active mutant
 2
 3  // mutated ''return a * b;'' statement using the ternary operator:
 4  float f(float a, float b) {
 5      return (MNR == 1 ? a + b :
 6              (MNR == 2 ? a - b :
 7              (MNR == 3 ? a / b :
 8              (MNR == 4 ? a % b : a * b))));
 9                         ~~~^~~~Invalid operands to binary expression
10  }
```

Execute Mutants. As a final step, all the mutants need to be executed. This can be done by running the executable for each valid mutant and only changing the environment variable. We again make the distinction between the reachable mutants, represented by mutants 1 to N, and the unreachable mutants represented by mutants X-Z in Figure D.2.

Expected Performance

For the mutant schemata approach, we estimate the performance via Formula D.2. The formula consists of three phases: the generate mutants, the compile mutants, and execute mutants phase. We included the generation of the mutants in the formula as this step is necessary and identical for all approaches but its impact should be negligible. We do not include the detection of unreachable mutants as it is not part of the schemata approach. We only need it to distinguish between the unreachable and reachable mutants. As the strategy removes the compilation overhead by only compiling the project once, instead of for each mutant, a drastic speedup is to be expected. Therefore, we no longer need to multiply the compilation time by the number of mutants. Note that the compilation time will be slightly longer, as there is more code that needs to be compiled. The execution part of the formula stays the same, as we still need to execute the test suite for each mutant. The test suite execution time for each mutant will, however, also be slightly longer as there is more code that needs to be executed due to the *if* statements of the ternary operator in order to activate the correct mutant.

$$\begin{aligned}
&t_{mutant\_generation} \\
&+ t^{schemata}_{compilation} \\
&+ t^{schemata}_{test\_suite\_execution} \quad *(unreachable\_mutants + reachable\_mutants)
\end{aligned} \tag{D.2}$$

Schemata Implementation

We implemented the mutant schemata technique using the ternary operator, the short-handed version of an *if* statement. We do note that the most common case for this implementation is the worst-case scenario. Most of the times no mutant in the statement will be activated, causing an evaluation of all the if statements before the original statement can be executed. We believe that this will create a fixed overhead per mutant, but we expect this overhead to be limited. The advantages of this approach is that it simplifies and streamlines the implementation of the schemata technique. It allows nesting the mutants inside the condition of an *if* statement. This can be seen in Listing D.5. The body of the if statement can also directly be mutated inside the expression. An alternative implementation would have been to use a switch case approach. However, this approach would have prevented us from nesting the mutants inside the condition of the if statement. Instead, we would have been forced to write the if statement inside the switch cases. This would have caused a code explosion by repeating the body of the if statement for each case, see Listing D.5. The body of the if statement could be mutated in the default case or by appending the cases of the mutated body to the existing switch.

Listing D.5: Mutating *if* Statements

```
// original
if (a > b) {/*body*/}

// mutated using ternary operator
if  (MNR == 1 ? a <  b :
    (MNR == 2 ? a <= b :
    (MNR == 3 ? a == b :
    (MNR == 4 ? a >= b : a > b)))) {/*mutated body*/}

// mutating using switch case
switch(MNR) {
    case 1:  if (a >  b) {/*body*/} break;
    case 2:  if (a >= b) {/*body*/} break;
    case 3:  if (a == b) {/*body*/} break;
    case 4:  if (a >= b) {/*body*/} break;
    default: if (a >  b) {/*mutated body*/} break;
}
```

Another advantage of the ternary implementation is that it causes no local scope and can thus mutate assign statements without additional analysis, unlike the switch case approach. On the other hand, to mutate an assign statement using the switch case approach, the variable would have needed to be defined outside the scope of the switch case.

D.3.3 Reachable Schemata

The previous strategy essentially eliminated the compilation overhead. This causes the test suite execution to become the most time-consuming part of the mutation analysis. While eliminating the completely unreachable mutants does speed up the mutation testing analysis, the reachable schemata strategy aims to use a more fine-grained approach to reduce most of the execution overhead. The reachable schemata strategy reduces the test suite scope to only those test cases which reach the mutant, effectively reducing the execution overhead. We timed the generation of the mutants, the compilation of all the mutants at once, and the execution of a reduced test suite for each of the mutants. We can then compare this to the unoptimised and mutant schemata strategies to gain insights into its speedups and potential overheads.

Exclude Invalid Mutants As the reachable schemata strategy is an optimisation based on the regular mutant schemata approach, we need to exclude the invalid mutants as well. This is done in the same way as with the mutant schemata approach.

Detect Reachable Mutants/Test In order to determine which mutants are reached by which test cases, we instrument the code base with a wrapper on the mutated locations. We store each mutated location a test run reaches. Running the instrumented code base for each test case provides us with lists of mutants that are reachable by each test case. This results in a reduced set of test cases which need to be executed for each mutant, resulting in a speedup without information loss.

The best results are obtained by using fine-grained test selection and running each test case individually. The speedups of this technique will depend on the test driver used for each project. Some test drivers might only be able to run individual modules instead of individual test cases. Speedups from module-grained test selection will be lower than from fine-grained test selection. Our projects were all compatible with fine-grained test selection on a test-by-test case basis.

Compile Mutants The compilation of the mutants is identical to the one for mutant schemata. We do note that an extra optimisation can be done by only inserting and compiling the reachable mutants in this phase. We, however, did not yet implement this optimisation in our proof-of-concept.

Execute Mutants As a final step, all the mutants need to be executed. Instead of running the entire test suite for each mutant, we now only need to run those test cases which reach the mutated locations. In Figure D.2 we see that mutant 1 is not reachable by the entire test suite, therefore no tests are executed for the mutant. For mutant 2, the second test case is not executed as that test case does not cover mutant 2.

Expected Performance

For the reachable schemata approach, we estimate the performance via Formula D.3. The formula consists of four phases: the generated mutants, the detect reachable mutants, the compile mutants, and the execute mutants phase. We included the generation of the mutants in the formula as this step is necessary and identical for all approaches but its impact should be negligible. We included the detection of the reachable mutants/test as it is a part of the reachable schemata approach in order to gather the reachable mutants to the test case relationship. We estimate its time impact at a single compilation and execution of the test suite. This is immediately compensated when there is more than one unreachable mutant. The compilation part again consists of a single compilation as it is an extension of the mutant schemata approach. For the execution part, only the reachable mutants are considered, together with a decoupling factor. This factor is a percentage based on the reduction in average mutants reachable by each test case. The more coupled a project is, the less effective this strategy will be. The less coupling there is (especially in combination with mocking), the more effective this strategy will be. From our measurements, we have seen that the average number of mutants reached per test is between 10 and 20% of all valid mutants. This implies that our technique can reduce the number of tests needed to execute per mutant between 5 and 10x. We therefore also expect a speedup between 5 and 10x.

$$\begin{aligned} & t_{mutant\_generation} \\ &+ (t^{schemata}_{compilation} + t^{schemata}_{test\_suite\_execution}) \\ &+ t^{schemata}_{compilation} \\ &+ t^{schemata}_{test\_suite\_execution} * reachable\_mutants * decoupling\_factor \end{aligned} \tag{D.3}$$

D.3.4 Split-Stream Mutation Testing

The split-stream mutation testing strategy reduces the execution overhead of mutation testing even further. Instead of letting each mutant initiate execution from the start of the program, each mutant is started from the mutation point itself. This can be achieved by exploiting the state-space information. Previous research has shown an average speedup of 3.49x of split-stream mutation testing over mutant schemata by mutating the LLVM IR [60].

We explain the overhead and general idea of the split-stream mutation testing approach using Figure D.3. Here we visualised the execution trace of the original, unchanged program, as a long vertical trace of states, each representing the execution of a single instruction. The trace of each mutant will look identical to the original mutant up until the mutated expression. In Figure D.3 we show mutant 4 with a mutation in the 4th expression, mutant 3 with a mutation in the 3rd expression, ... The trace will only start to deviate from the original trace after the mutated expression. All of the states up until the mutated state are in fact redundant, as we already know them from the execution of the original program. We highlighted these redundant states and expressions in red in Figure D.3.

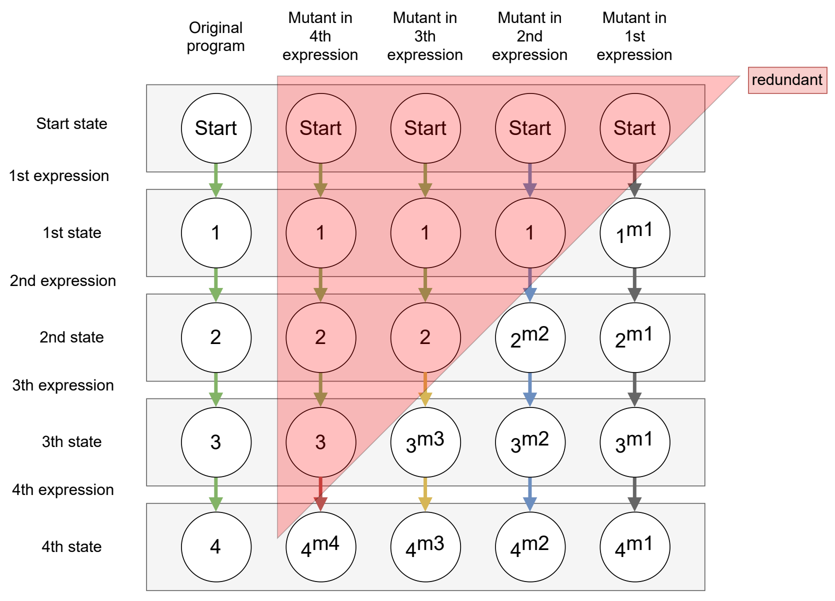


Figure D.3: Trace of Original and Mutated programs

Exclude Invalid Mutants As this strategy is an optimisation from the regular mutant schemata approach, we need to exclude the invalid mutants as well. This is done identically as with the mutant schemata approach.

Compile Mutants The compilation of the mutants is identical to the one for mutant schemata. However, additional code needs to be instrumented in the code base in order to exploit the state space and start the mutants from their mutation point instead of from the start of the program.

Execute Mutants In order to exploit the state space, we start the split-stream mutation analysis only once instead of for each mutant as with mutant schemata. We will explain the split-stream mutation testing process using Figure D.4.

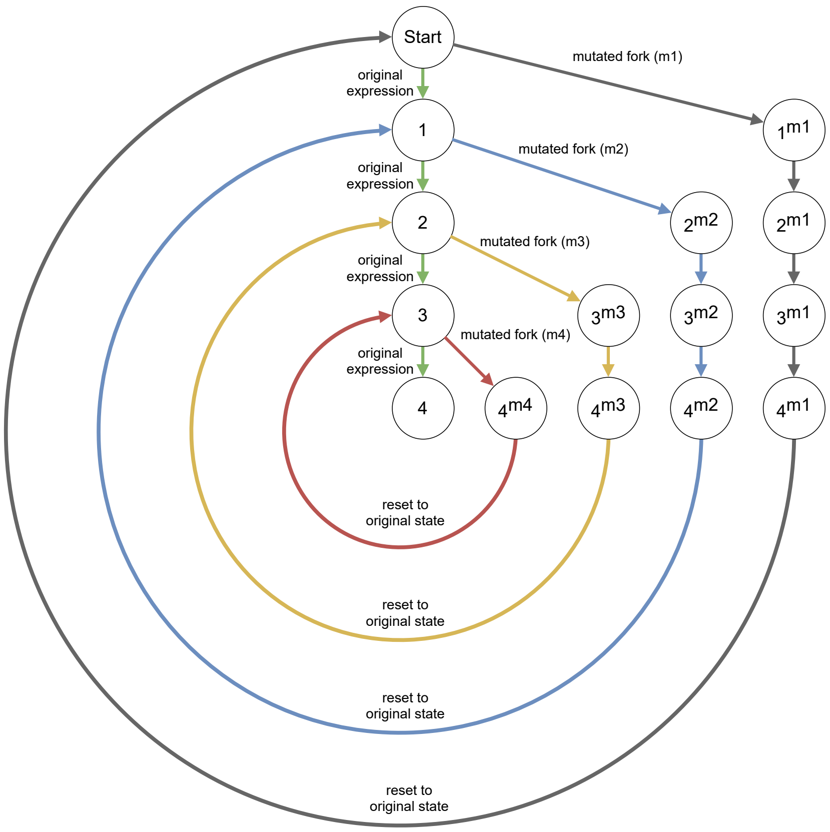


Figure D.4: Split-Stream Mutation process

When we start the split-stream mutation analysis, no mutant is active. The first mutant the program will encounter will not yet have been activated, so the program will fork the entire program and pause the original one. The forked program will be in the exact same state as the original program, essentially this is a duplicate of the state-space. In the forked program, we activate the encountered mutant. We let the forked program execute, only taking into account the active mutant and store its results at the end of its execution. We then continue our first program, until it encounters another mutant that it has not yet activated. Here the process is repeated, a fork is created, it is executed, and its results are stored. This goes on until the original program reaches its end state.

Like with the mutant schemata strategy, our proof-of-concept tool needs to decide what needs to happen at each mutation position. Instead of only deciding whether or not a mutant needs to be active or not, it now also needs to decide *when* it needs to fork the process. For this, we instrumented additional code at each mutation point. We explain the general concept using the C++ example in Listing D.6.

When we start our program with all the mutants in it, no mutant will ever be activated in it. The original program is only responsible for forking newly encountered mutants and storing their results.

When a mutant is encountered in the main program, e.g. on line 16, then it will continue to line 7, where it will verify whether or not it already encountered that mutant. If it did, it will continue the program, otherwise, it will fork the program and wait for the forked program to finish, on lines 8 and 9. The forked program will only take into account the active mutant and run till its end. When the forked program encounters a mutant, it will verify if it is the mutant that the process was forked for, if so it will execute it, otherwise it will execute the original code. The forked process will not create additional forks. The main program will then store the result (exit code) of the forked program, add the mutant to the already executed list and continue until it encounters a new mutant, on lines 9 and 10.

Listing D.6: Split-Stream Mutation Testing Example

```
 1 extern list<int> MNR_list;  // already activated mutants
 2 extern int MNR;             // active mutant
 3
 4 bool split(int mnr) {
 5     if (/* this is the fork */) { return mnr == MNR; }
 6     else { /* this is the main program */
 7         if (MNR_list.contains(mnr)) {return false;}
 8         else { /* fork, and set MNR to mnr in forked process*/ }
 9         /* wait for fork to complete and store results */
10         MNR_list.insert(mnr);
11     }
12     return false;
13 }
14
15 float f(float a, float b) {
16     return (split(1) ? a + b :
17            (split(2) ? a - b :
18            (split(3) ? a / b : a * b)));
19 }
```

The split-stream mutation testing approach naturally ensures that unreachable mutants are not executed. The approach will never create forks for mutants that it will not reach. By executing each of the tests separately, only those mutants reachable by the test case will be executed for that test. This is similar to the reachable schemata approach, with the added benefit that there is no separate compilation and/or test suite execution run necessary in order to extract the reachable mutants.

As we start the mutants from their mutation points, we now also need to ensure that all of the project dependencies, e.g. files and databases, are in their correct state.

As this will likely require specific project knowledge and will differ for each project, we built in hook functions which can be specialised for each project. Additionally, we built in support to automatically reset local files to the state they were in for each of the mutation points.

Expected Performance

For the split-stream mutation approach, we estimate the performance via Formula D.4. The formula consists of three phases: the generate mutants, the compile mutants, and the execute mutants phase. We included the generation of the mutants in the formula as this step is necessary and identical for all approaches but its impact should be negligible. We no longer need to detect the reachable mutants/test as it is inherent to the split-stream approach. The compilation part again consists of a single compilation, which should be slightly longer than the compilation of the schemata approach. The execution time will be impacted negatively as the forking of the process will take some time. As the forking process works on a copy-on-change principle, only the memory that is changed will be copied. This ensures that the impact of the forking is kept at a minimum. The benefit of this strategy is that we no longer need to execute the redundant code. A mutant that is located at the very end of the execution will now only take very little time to execute. This is in contrast to the original mutation testing where it would have needed to start its execution from the very beginning. Mutants located in the front will only have a small improvement in their execution time. In general, one could thus assume that this strategy would contribute to an additional 2x speedup. Similar to the reachable schemata approach, executing the test suite on a test-by-test case will yield the best results. Hence, we have the same decoupling factor.

$$\begin{aligned}
&t_{mutant\_generation} \\
&+ t^{split\_stream}_{compilation} \\
&+ t^{split\_stream}_{test\_suite\_execution} * reachable\_mutants * decoupling\_factor/2
\end{aligned} \tag{D.4}$$

D.4 Experimental Set Up

This paper presents a feasibility study, investigating to which extent the Clang front-end and its state-of-the-art program analysis facilities allow to implement existing strategies for mutation optimisation within the C language family. We measure the speedup from two perspectives (compilation time and execution time) assessing four optimisation strategies. This gives rise to the following research questions:

- *RQ1: How much speedup can we gain from mutant schemata?*
- *RQ2: How much speedup can we gain from split-stream mutation testing?*
- *RQ3: How much speedup can we gain from eliminating invalid mutants?*
- *RQ4: How much speedup can we gain from eliminating unreachable mutants?*

To answer these research questions, we need to measure the impact of our optimisations, which delays we introduce, how much of the overhead we eliminate, and more importantly where these strategies can be applied.

D.4.1 Cases

To investigate the strengths and weaknesses of the Clang-based optimisation strategies, we validate our proof-of-concept tool on four open-source C++ libraries and one industrial component. These cases cover a wide diversity in size, C++ language features used, compilation times, and test execution times as shown in Table D.3. To allow other researchers to reproduce our results, we refer to the latest commit id of the version of the project we used for our analysis.

[5] http://cloc.sourceforge.net

Table D.3: Results: Project Details

	TinyXML2	JSON	Google Test	CppCheck	Saab Case
Commits	1,052	4,312	3,840	25,309	
Contributors	78	213	100	340	
GitHub stars	3.9k	28.4k	24.9k	3.9k	Confidential
LOPC	3,542	10,955	32,630	98,171	
LOTC	1,885	26,586	28,064	159,780	
Test Cases	1	88	61	3,745	
Compilation time	1.12s	3m 52.88s	6m 31.34s	5m 49.68s	3m 08.52s
Test suite run time	0.11s	14m 10.76s	15.01s	17.04s	2.63s
Generated Mutants	1,038	3,764	4,488	61,007	
Excluded Mutants (Const, Constexpr or Templated Mutants)	185	3,254	1,755	5,591	Confidential
Considered Mutants	853	510	2,733	55,416	
Mutants/LOPC	0.241	0.047	0.084	0.564	0.233
Valid Mutants	716	333	2,498	54,643	
Invalid Mutants (killed by compiler)	137	177	235	773	
Completely Unreachable Mutants	36	0	144	5,712	Confidential
AVG Tests/	0.95	8.46	11.91	402.96	
Reachable Mutant	94.97%	9.61%	19.53%	10.76%	
Survived Mutants	331	33	1,028	21,291	
Killed Mutants (excl. timed out)	211	300	1,419	24,639	Confidential
Timed out	138	0	51	8,713	

LOPC = Lines of Production Code; LOTC = Lines of Test Code. LOPC (incl. include files) calculated using cloc[5] (excl. newlines and comments)

In the first block of the table, we list general details about the project, number of commits, contributors, Lines Of Project Code (LOPC), Lines Of Test Code (LOTC), and the number of test cases the project has.

The second block list the compilation time of the project and the test suite execution time. Projects with a longer compilation time might benefit more from a mutant schemata strategy than projects with a longer test suite execution time.

In the third block, we list how many mutants we generated for each project. As our mutant schemata optimisation cannot yet work with mutants in so-called const-expression and/or templates, we excluded these mutants for a fair comparison. We also listed the number of mutants per line of production code (LOPC) which indicates how densely or sparsely packed the mutants are. A project with a high density of mutants might introduce delays specific to the optimisation strategy.

In the fourth block, we list the number of valid mutants and the number of invalid mutants, i.e. mutants that are killed by the compiler. We also list the number of mutants that are not reachable by the current test suite. Finally, we list the average amount of test cases that actually reach a valid mutant. We also listed thin in percentages to show how much of the total test cases are considered for each reachable mutant. The reachable mutant schemata strategy uses this information to its advantage to reduce the mutant execution time.

In the last block, we list the number of survived mutants and the number of killed mutants.

TinyXML2

`https://github.com/leethomason/tinyxml2`	commit id: ff61650517cc32d524689366f977716e73d4f924

TinyXML2 is a simple, small, efficient, C++ XML parser that can be easily integrated into other programs. It represents our small-sized project with 3,542 lines of production code and 1,885 lines of test code (without empty lines and comments). Even though the project is small, we generate 1,038 mutants for it. We choose this project deliberately because of its short compilation and execution time (1.12s and 0.11s respectively), as it represents a worst-case scenario for any overhead introduced by the optimisation strategy.

JSON

`https://github.com/nlohmann/json`	commit id: 7c55510f76b8943941764e9fc7a3320eab0397a5

JSON is a special case as the entire source code consists of a single header file. This means that any changes to that file will cause a complete rebuild of the entire project, including virtually all test files. Furthermore, all valid mutants are reachable by the test suite. This means that there will be no speedup from detecting the unreachable mutants.

The test suite of JSON consists of 88 tests with a runtime of 14 minutes. Two of the tests however cause the majority of this time, i.e., test 12 with 1 minute and test 80 with 11 minutes execution time. By limiting the number of mutants to only those mutants that are reached by these tests, the total runtime can be reduced drastically.

Google Test

`https://github.com/google/googletest`	commit id: f2fb48c3b3d79a75a88a99fba6576b25d42ec528

Google Test represents our medium size project with 32,630 lines of production code and 28,064 lines of test code. It is a widely used framework for testing C++ code, so one could expect it to be fairly reliable. This is confirmed by our mutation coverage, from the 2,498 valid mutants, only 144 (5.8%) are unreachable by the test suite.

CppCheck

`https://cppcheck.sourceforge.io`	commit id: 8636dd85597acdc1560f7e0bd364c94851bec3b9

CppCheck is a static analysis tool for C/C++ programs. It aims to detect bugs, undefined behaviour, and dangerous coding constructs. It has the most mutants per line of production code and will thus also have the largest negative impact on the duration of its test suite execution. It is the biggest of the projects we analysed with the highest number of mutants. We would like to note that this project has many configurable macros. We ran the project without changing the defaults. Running the project with different macro configurations can lead to different results. This can even increase or decrease the number of unreachable mutants.

Saab Case

`https://saabgroup.com/about-company/organization/`
`business-areas/`

The Saab Case represents the project from our industrial collaboration. The company develops safety-critical systems and must adhere to 100% MC/DC testing (Modified Condition/Decision Coverage, the coverage criterion adopted for the highest Design Assurance Level (DAL) in accordance to the RTCA-DO178B/C standard). This means that their project is well tested. Due to the project being classified as confidential, some information has been left out and marked *Confidential* in the coming sections. However, the information related to the speedup caused by the use of mutant schemata can be disclosed.

D.4.2 Hardware Set-up

We used the same infrastructure for the analysis of the selected open-source projects. We used an Intel(R) Core(TM)2 Quad Q9650 CPU, with two 4GB (Samsung M378B5273DH0-CH9) DDR3 RAM modules (for a total of 8GB) and a 250GB Western Digital (WDC WD2500AAKX-7) hard drive. The PC was running Ubuntu 18.04.1 LTS (GNU/Linux 4.15.0-29-generic x86_64). Using an SSD will drastically influence the compilation times and negatively impact the total speedups. Using more RAM and/or a faster CPU might influence the compilation time and/or execution time, this however is presumed to be marginal.

The industrial project of Saab Aeronautics ran on their build server. Some of their results could be slightly impacted by background services.

D.5 Results and Discussion

Before we measure the speedup induced by the different optimisation strategies implemented using the Clang front-end, we first analyse the time spent in each of the optimisation phases. The individual timing results can be found in Table D.4. In essence, this timing information allows us to assess the significant terms in the performance estimation formulas for the unoptimised configuration (Formula D.1 (p. 83)); the mutant schemata (Formula D.2 (p. 84) and Formula D.3 (p. 87)) and the split-stream mutation testing (Formula D.4 (p. 90))

D.5.1 Individual Timings

Generate Mutants. The first block of Table D.4 confirms that the generate mutants phase is negligible. This phase is necessary and identical for all our optimisation techniques and corresponds to $t_{mutant\_generation}$ in Formula D.1 to 4. The four open-source cases illustrate that this phase is orders of magnitude faster than any of the following steps and less than the original compilation time of the projects as listed in Table D.3.

Detect (Un)Reachable Mutants. In the second block of Table D.4, we listed the detection times of the (un)reachable mutants. For the unoptimised and schemata technique, we detected which mutants were reached by the test suite. The reachable schemata approach is more fine-grained and detects which mutants are reached on a test-by-test case. We estimated this phase at $\left(t^{schemata}_{compilation} + t^{schemata}_{test\_suite\_execution}\right)$, a single compilation and test suite execution in Formula D.3. The four open-source cases and the industrial case illustrate that this phase is also orders of magnitude faster than any of the following steps.

As the split-stream approach inherently only executes reachable mutants, it does not have a detection delay.

Table D.4: Results: Individual Timings

Strategy	Mutants	TinyXML2	JSON	Google Test	CppCheck	Saab Case
Generate Mutants & validation:						
All		0.09s	0.64s	3.79s	2m 47.15s	Confidential
Detect (Un)Reachable Mutants:						
Unoptimised	Reachable /test suite	0.11s	14m 12.09s	7m 11.39s	7m 51.60s	2.63s
Schemata						
Reachable Schemata	Reachable /test case					
Split-stream		0s	0s	0s	0s	0s
Compile Mutants:						
Unoptimised	Reachable	7m 17.49s	21h 45m 46.62s	40h 21m 45.49s	3d 18h 52m 39.52s	Confidential
	Unreachable	22.79s	0s	2h 19m 35.25s	10h 36m 20.31s	
	Invalid	50.83s	3h 05m 23.74s	3h 05m 24.35s	1h 01m 41.15s	
Schemata	Valid	1.23s	3m 54.68s	6m 36.04s	7m 07.39s	3m 02.63s
Reachable Schemata						
Split-stream		1.28s	3m 57.27s	6m 41.85s	7m 13.91s	N.A.
Execute Mutants (excluding timed out):						
Unoptimised	Reachable	1m 05.60s	17h 41m 49.05s	9h 49m 36.26s	2d 05h 41m 30.47s	Confidential
	Unreachable	4.10s	0s	36m 01.25s	1d 03h 02m 43.46s	
Schemata	Reachable	1m 21.84s	17h 43m 28.57s	9h 50m 30.45s	4d 22h 16m 45.61s	
	Unreachable	4.81s	0s	36m 04.84s	2d 11h 32m 05.66s	
Reachable Schemata	Reachable /test case	1m 13.54s	1h 01m 29.01s	1h 46m 36.16s	9h 27m 30.01s	
Split-stream		DNF				
Technique Execution Overhead:						
Schemata vs Unoptimised	Reachable	17.44%	0.16%	0.15%	120.29%	0.11%
	Unreachable	19.95%	N.A.	0.17%	120.13%	
Timed Out Mutants:						
Unoptimised		27.60s	0s	26m 00s	2d 12h 30m 25s	Confidential
Schemata					5d 01h 00m 50s	
Reachable Schemata						
Split-stream		DNF				

Compile Mutants. The third block of Table D.4 confirms that the compilation phase takes up a significant amount of time. Additionally, the compilation of the invalid and unreachable mutants is considerable.

We estimated this phase for the unoptimised approach at $t_{compilation} * (reachable\_mutants + unreachable\_mutants + invalid\_mutants)$ in Formula D.1.

By moving from the unoptimised approach to the schemata approach, we only need to compile once instead of for all mutants, reducing the compilation time to $t^{schemata}_{compilation}$ from Formula D.2 where the multiplication is removed. This essentially removes the compilation overhead as is confirmed by the four open source cases in Table D.4. We can also see that the compilation is only slightly longer than the original compilation, as seen in Table D.3. This is due to the fact that the injected mutants increase the code base, but this is limited to each function scope, therefore leaving the linking part of the compilation untouched. However, when we introduce more advanced mutation operators and mutate the aforementioned const expressions, we likely need to introduce different functions, causing the compilation time to increase further. Still, we can expect the compilation time to remain drastically decreased from an unoptimised traditional approach. This is also the case for the split-stream mutation testing approach, which in its turn is slightly longer compared to the mutant schemata one, as it has internal functions to regulate the activation and forking of the mutants.

Execute Mutants. The fourth block of Table D.4 contains the execute mutant phase excluding the timed out mutants. Here we can see that the execute mutant phase also takes up a significant amount of time. This was expected as we estimated its duration at $t_{test\_suite\_execution} * (reachable\_mutants + unreachable\_mutants)$ for the unoptimised and schemata approach in Formula D.1 and D.2. The first observation we can make is that the unreachable mutants can have a minimal to large impact on the mutant test execution. The stronger a test suite is, the fewer mutants will be completely unreachable. We see no impact in the JSON project as it has no unreachable mutants. We see the most impact in the CppCheck project. Here, the test execution time of the unreachable mutants for the *unoptimised and mutant schemata* approach is half of the reachable ones. It is possible that running the project with a different configuration leads to a different number of unreachable mutants.

Our second observation is that the execution time of the reachable schemata approach is drastically shorter than that of the schemata and unoptimised approach, even when compared to only the reachable mutants. Instead of executing the entire test suite for each mutant, the reachable schemata approach only executes those test cases that reach the specified mutant. A mutant that survives will have fewer test cases executed compared to the mutant schemata and unoptimised approach. It also stands to reason that mutants that are detected are detected faster as the test cases that do not reach the mutant are not executed. We estimated its duration at $t^{schemata}_{test\_suite\_execution} * reachable\_mutants * decoupling\_factor$ in Formula D.3. In Table D.4 we can see that the decoupling factor varies from project to project. This factor can roughly be represented by the average amount of tests that will be executed per mutant, which we listed in Table D.3. TinyXML2 has only 1 test, so there is no additional speedup to be gained from the reachable schemata approach. The JSON project has the most speedup from this approach as it only needs to execute 9.61% of the test cases per mutant, this is followed by the CppCheck project at 10.76% and the Google Test project at 19.53%.

Our last observation is that we, unfortunately, could not gather data regarding the test suite execution of split-stream mutation testing on the projects under analysis. We, therefore, labeled its result as DNF. While we build in support to reset local files to a specific state using a local git repository, and built in hook functions to reset more advanced dependencies like running, or even off-site, databases, we could not get this to work reliably for the tested projects. Our projects all relied on open IO-streams which we cannot reset unless we reimplement or overload these functions. For our projects, this meant that eventually, the IO files got corrupted and/or the file offset pointer. We did not search for a solution as while this strategy offers an expected speedup of 2x, the strategy is difficult to incorporate with external dependencies such as databases as it requires in-depth knowledge of the system. We, therefore, do not recommend using the strategy, at least not as a first option.

Technique Execution Overhead. In block four we can see that the execution time for both the unreachable and the reachable mutants is increased by using the schemata technique compared to the unoptimised technique. This is due to the additional run-time instructions that ensure the correct activation of the mutants. We listed the execution overhead introduced by the optimisation techniques in the execute mutant phase in percentages in the fifth block of Table D.4. We can see that the percentual overhead between the unreachable and reachable mutants is, as expected, very close. As there are no unreachable mutants for the JSON project, no overhead can be calculated for it. We can also see that there is a limited overhead for the reachable mutants in the JSON and Google Test projects of 0.16 and 0.15%. The TinyXML2 project is impacted more by 17.44%. This increase in overhead can be attributed to the increase in the number of mutants per LOPC. Where this was only 0.047 and 0.084 for JSON and Google Test, it is 0.241 for TinyXML2. The more mutants there are per LOPC, the more if statements there are to verify which mutant needs to be executed. Mathematically dense programs will suffer more from the schemata implementation using the ternary operator. Switching to a switch-case implementation would reduce the impact. The CppCheck case has, with 0.565 mutants per LOPC, the highest number of mutants per LOPC. We thus also expected a higher impact on the execution time of the project. We measured an overhead of 120.29%. This overhead can be attributed to the many if statements that are introduced per statement. Changing the activation of the mutants from if statements to switch-case-based should reduce this overhead.

Timed Out Mutants. For many of the projects, the test time caused by timed-out mutants is fairly limited. This can be seen in the last block of Table D.4. However, for the bigger, and longer running projects, this does become a significant amount of the mutation analysis time. This is especially true for the CppCheck project as the timed-out mutants take up 93% of the reachable schemata mutation analysis. We manually set a fixed timeout time for each of the projects. The time out is identical for each of the optimisation approaches except for the CppCheck project. The introduced overhead caused an additional delay for the schemata-based approaches. Here we double the time-out time. The time-out time can be reduced specifically for the reachable mutant schemata strategy. Here we execute each test case individually instead of the entire test suite. We should thus set an individual time out for each of the test cases. These time-outs will thus be much shorter, resulting in a reduced detection time when a mutant times out due to, e.g., an infinite loop.

D.5.2 Complete Mutation Analysis

In this section, we look at the impact of the optimisation strategies in relation to the complete mutation analysis whilst answering the research questions. Their timings and speedups can be found in Table D.5.

RQ1: How much speedup can we gain from mutant schemata?

The *schemata approach*, which virtually eliminates the compilation overhead, drastically speeds up the mutation analysis in most cases. As we have seen in Section D.5.1, mutant schemata can reduce the compilation time down to almost the original, single, compilation time of the

Table D.5: Results: speedups

Mutants	TinyXML2	JSON	Google Test	CppCheck	Saab Case
Unoptimised:					
Considered	10m 12.42s	1d 18h 33m 00.05s	2d 09h 14m 27.64s	11d 06h 50m 50.51s	
Valid	9m 17.58s	1d 15h 27m 36.31s	2d 05h 33m 02.04s	10d 02h 46m 25.90s	
(vs considered)	1.10x	1.08x	1.07x	1.12x	Confidential
Reachable	8m 52.25s	1d 15h 45m 50.43s	2d 02h 44m 36.93s	8d 13h 15m 13.74s	
(vs valid)	1.05x	0.99x	1.06x	1.18x	
Schemata:					
Valid	1m 50.77s	17h 47m 23.89s	10h 59m 15.12s	12d 10h 59m 35.81s	
(vs unopt. valid)	5.03x	2.22x	4.87x	0.81x	Confidential
Reachable	1m 47.43s	18h 05m 38.01s	10h 30m 21.67s	9d 23h 59m 35.81s	
(vs unopt. reachable)	4.95x	2.20x	4.83x	0.86x	5.16x
(vs schemata valid)	1.03x	0.98x	1.05x	1.25x	Confidential
Reachable Schemata:					
Reachable/test	1m 43.92s	1h 23m 38.45s	2h 26m 27.38s	5d 10h 46m 06.16s	
(vs unopt. considered)	5.89x	30.52x	23.45x	2.07x	
(vs unopt. reachable)	5.12x	28.52x	20.79x	1.57x	Confidential
(vs schemata reachable)	1.03x	12.98x	4.30x	1.83x	
(vs valid schemata)	1.07x	12.76x	4.50x	2.29x	

project. Here we see speedups of up to 5.03x for the valid mutants, and up to 4.95x for the reachable mutants compared to the unoptimised approach, as listed in Table D.5. JSON has a speedup of 2.22x due to a high proportion of test suite execution time versus its compilation time. The CppCheck project obtained a speedup 0.81x due to the increase in mutant execution time, attributed to the techniques implementation in combination with the high number of mutants/LOPC. In the previous section we saw that there was a limited impact on the execution times of the mutants in projects with a low number of mutants per line of production code (less than 0.1: JSON and Google Test), projects with a high number of mutants per line of production code (0.24 and 0.56: TinyXML2 and CppCheck) suffered from increased execution overheads which limited its speedup potential. We proposed mitigating the overhead issue by using *switch* statements instead of *if* statements for mutant selection. However, we leave the implementation and evaluation of this mitigation as future work. But, we then expect to also see a speedup for this project.

We measured a speedup of 5.16x for the industrial Saab project for the mutant schemata approach compared to a traditional approach when excluding the completely unreachable mutants. For the other projects we see that the speedups for this optimisation are slightly smaller than with the unreachable mutants. This is as expected as more time (compilation and execution time) is saved excluding the unreachable mutants in an unoptimised approach then in a schemata approach (execution time).

With the TinyXML2, Google Test and the industrial Saab project, the mutant schemata technique obtains a speedup between 4.87x and 5.16x. For the other cases the speedup was less eminent. JSON has a speedup of 2.22x due to a high proportion of test suite execution time versus its compilation time. CppCheck obtained as speedup of 0.81x due to due to the increase in mutant execution time, attributed to the techniques implementation in combination with a high number of mutants/LOPC. The obtained speedup from a mutant schemata technique depends on two factors: (a) the proportion of the project compilation time versus its execution time; (b) the number of mutants per line of production code. The latter is implementation specific and its impact can be reduced by changing the activation of the mutants from if statements to switch-cases.

RQ2: How much speedup can we gain from split-stream mutation testing?

For *split-stream mutation testing* we expected an additional speedup of approximately 2x over mutant schemata. Unfortunately, we could not gather data regarding the test suite execution on the projects under analysis due to IO dependent issues. While we build in support to reset local files to a specific state using a local git repository, and built in hook functions to reset more advanced dependencies like running, or even off-site, databases, we cannot recommend this optimisation technique as it cannot easily be incorporate with external dependencies such as databases as it requires in-depth knowledge of the system.

In theory this strategy could yield a speedup of a factor 2, but in practice applying the strategy proves to be too demanding. Reverting to the internal state demands too much specific knowledge about the design of the system under test, especially in the case data is stored in external databases and filesystems.

RQ3: How much speedup can we gain from eliminating invalid mutants?

In Table D.5 we see that for the *unoptimised approach* a speedup between 1.07x to 1.12x is achieved by excluding the invalid mutants. This speedup is attained by a reduction in the compilation time as invalid mutants are mutants that cause compilation issues and can thus not be executed.

Compiler-integrated techniques like mutant schemata and split-stream for the C language family come with an important drawback: the tight integration with the compiler. Since all mutants are injected simultaneously, the resulting program must compile without any errors. Invalid mutants are not acceptable, since they prevent the compilation (and the execution) of the mutated program. This is especially challenging for statically typed languages with many interacting features and unforgiving compilers (C, C++, . . .). Since Clang has access to all of the project's type-information, we can programmatically ensure that any of the created mutants are statically correct. We can therefore not calculate a speedup for the invalid mutants as their exclusion is a requirement for the schemata techniques.

Compared to an *unoptimised* approach the reduction in the compilation overhead leads to a speedup by a factor between 1.07x and 1.12x. This is not all that much, but excluding invalid mutants is a necessary prerequisite for the mutant schemata strategy discussed under RQ1.

RQ4: How much speedup can we gain from eliminating unreachable mutants?

In Table D.5 we see that for the *unoptimised approach* a speedup between 1.05x to 1.18x is achieved by excluding the completely unreachable mutants, except for the JSON project as it has no unreachable mutants This speedup stays approximately the same with the *mutant schemata* approach where the speedup is between 1.03x to 1.25x by excluding the completely unreachable mutants, except for the JSON project as it has no unreachable mutants. This speedup, however, will depend on the coverage of the test suite. A test suite with low coverage, and thus reaching fewer mutants will yield a higher speedup by excluding them.

However, the *reachable schemata* technique goes a step further by not only excluding completely unreachable mutants but by also excluding the test cases which do not reach the mutant on a mutant by mutant case. This gives an additional speedup between 1.83x and 12.98x over the schemata technique that excludes completely unreachable mutants. This provides a speedup between 2.29x and 12.76x over the normal schemata technique, except for the TinyXML2 project, as it only has a single test case. Further speedups are possible by optimising the time-out function that e.g. detects mutants stuck in infinite loops. This can be achieved by specifying a time-out for each test case instead of a time-out for the global test suite.

Compared to an *unoptimised* approach, excluding the completely unreachable mutants provides a speedup between 1.05x to 1.18x. If we go one step further (also excluding the test cases which do not reach the mutant on a mutant by mutant base) we achieve an additional speedup between 1.83x and 12.98x.

D.6 Limitations and Lessons Learned

In this section, we will derive the limitations and lessons learned geared toward the mutation testing community.

Mutant schemata for C++ can bring a performance improvement of an order of magnitude by eliminating the compilation overhead. However, mutant schemata for C++, and all strongly typed languages, require the ability to extract extra information to ensure that none of the mutants causes compilation errors as all mutants need to be compiled at once. We achieved this by using Clang, allowing us to extract the statically available information and to create the mutants in the actual source code. Mutant schemata and the additional speedup strategies also require instrumentation of the source code. For this, we also relied on Clang. Doing our work we learned 10 lessons, which we list below.

Type Safety. Some challenges occur when we try to implement mutant schemata for statically typed programming languages like C++. First and foremost, the mutated program must be syntactically correct and no type errors should occur. This means that every mutated statement should be valid. We cannot generate mutants like “string − string” or “float % int”. Classes can implement or omit operators like “+” and “−” further complicating the matter. Clang allows us to access all the statically available information of the project and to verify if a mutated statement is syntactically correct without the need to compile the complete project.

While we currently have this working for binary expressions, we have not investigated how this needs to be done for less straightforward mutations such as Access Modifiers Change (AMC), where e.g. the public access label is changed to private. While we believe this can be done using Clang, we envision that the analysis time for such a change will be longer compared to the analysis of a binary expression.

Ternary Operator. Using the ternary operator also somewhat limits the mutation kinds we can use, as all mutants in the expression need to be of the same type. We cannot have a char pointer on the *iftrue* side and an unsigned int on the *iffalse* side. Fortunately, virtually all mutations on a single expression will have the same type or can be dynamically cast to that type.

Mutation Operator Support. Currently, for our proof-of-concept tool, we have support for Relational Operator Replacement (ROR), Arithmetic Operator Replacement (AOR) and Logical Connector Replacement (LCR). Our proof-of-concept tool can be easily extended to support other binary mutation operators and unary operators. However, other mutation operators, like the previously mentioned Access Modifiers Change (AMC) might need more work.

Const, constexpr, and templates. The driver for mutant schemata relies on information from outside the program to control the activation of the mutants by setting the MUTANT_NR variable. The MUTANT_NR variable is initialised at runtime and will thus never be const. This means that we cannot use the variable inside const and constexpr functions, as these functions are evaluated at compile time and the MUTANT_NR value cannot be known at compile time. An example of this can be seen in Listing D.7.

Listing D.7: Invalid Schemata from const

```
const float a = 1.2;
const float b = 2.0;
// original statement
const float r = a * b;

// mutated statement
extern int MNR;      // driver for active mutant
const float r = (MNR == 1 ? a + b :
                (MNR == 2 ? a - b :
                (MNR == 3 ? a / b : 1.2 * 2)));
                       ~~~^~~~Invalid schemata MNR cannot be know at
                          compile time
```

This means that we cannot mutate const and constexpr in the same way as non-const functions. This includes type definitions (e.g. *using* ...), template arguments, static_asserts, etc.

For now, we choose not to implement support for these kinds of mutations. We do however generate and verify these for a standalone mutation, but using them in a single compilation for mutation testing using mutant schemata and/or split stream requires additional logic. This can be implemented by creating separate values and/or functions for each mutated operation and selecting the correct one everywhere in the project where they are used (e.g. const val becomes const val_0, const val_1, const val_2, ...). This will drastically increase the size of the code base and the compiled binary. Additionally, this will reduce the readability of the code, however, normal developers should not see this.

Switch-Case. In Section D.3.2 we explained the advantages of using the ternary operator for a mutant schemata approach. We envisioned that there would be a fixed overhead cost for the technique per mutant, but that this overhead would remain limited. In our results we saw that this was only the case for projects with a low to medium number of mutants per LOC. For projects where the number of mutants per line of production code is high, such as math-heavy projects, the ternary implementation causes too many additional evaluations of the if statements to reach the correct mutant. The most common case where no mutant in the statement is activated is the worst-case scenario, as all the if statements need to be evaluated before the original statement can be executed. In the simplistic example in Listing D.8, we can see that an "$a * b * c$" statement has six mutations. This means that for every mutant outside of that expression, six if statements would need to be evaluated. By switching to a switch-case implementation like in Listing D.8, only the switch needs to be evaluated and the offset for which case to jump to needs to be calculated. This would save many instructions especially when there are many instructions on a single line of code. This would have greatly benefited performance in the CPPCheck case.

To reduce the overhead of the schemata strategy, we recommend using the switch-case implementation in larger projects, despite the implementation challenges that need to be overcome. We described some of these challenges and tradeoffs in Section D.3.2, including code explosion and the scoped nature of the switch case causing the need to initialise variables outside the switch. Because of these challenges, specific research will need to be performed to understand where and how switch cases should be used, and how to deal with its tradeoffs. We therefore deem this as out-of-scope for the current research, but as something that needs to be investigated in future work. It is possible that if statements and switch-cases should be used together to achieve an optimal speedup.

Listing D.8: Mutant Schemata using Switch Case

```
extern int MNR;      // driver for active mutant
float f(float a, float b, float c) {
    switch (MNR) {
        case 1:  return (a * b) + c;
        case 2:  return (a * b) - c;
        case 3:  return (a * b) / c;
        case 4:  return a + (b * c);
        case 5:  return a - (b * c);
        case 6:  return a / (b * c);
        default: return a * b * c;
    }
}
```

Test per test case vs per module. Our program extracts the reachable mutants per test case or per module for the program. The best results are obtained by using fine-grained test selection and running each test case individually instead of module-grained. All our projects are compatible with the fine-grained test selection. Testing per test instead of per module reduces the AVG number of tests per mutant. We verified this for the CPPCheck project. Here the AVG tests/mutant for the fine-grained test selection is 403, while it is 812 for the module-based one. The additional 409 tests selected by the module-based approach do not cover the mutant and will never be able to kill the mutant. Given a random distribution of these tests, we can assume that twice the amount of test cases will be executed per mutant for the CPPCheck project. If all test cases take approximately the same amount of time, the reachable schemata technique per test is twice as fast as the per module. Naturally, the AVG tests/mutant for the fine-grained test selection or the module based one, and hence the speed difference, will depend on the project itself.

The CPPCheck program implemented its own testrunner to enable running the tests per test instead of per module. If you use ctest, then traditionally, tests are added using the *add_ test* command to a specific test module. Each test module can be run separately, but you cannot run a specific test within a test module without running the other tests from that module. By switching from the *add_ test* command to *gtest_ add_ tests* and/or *gtest_ discover_ tests*, one can run each test individually. This means that each test will run slightly slower, as a new test environment will be created for each test instead of for each module. This will however be greatly offset by the reduction in the number of mutants that will be selected to run for each test.

Multi-threading. Traditional, unoptimised mutation testing and mutant schemata inherently support applications that utilise multiple threads as they run from start to finish. Split-stream mutation testing on the other hand does not. If we 'start' the first mutant after a secondary thread was spawned, our mutant will likely run to the end of the program and wait for the secondary thread to finish. A second mutant 'started' from the same location will expect the secondary thread to be there, but as it is already closed, the second mutant will not run correctly. Supporting multi-threaded programs for the split-stream approach will require additional development to ensure that not only the main thread will be forked, but also the existing threads.

External Dependencies. Traditional, unoptimised mutation testing and mutant schemata are highly likely to support external dependencies. This is due to the fact that external dependencies are a common phenomenon in Continuous Integration settings. After the test suite has run, the external dependencies, like input/output files and databases, need to be reset for the next run of the Continuous Integration. The support for this is usually baked into the used build systems of the projects (e.g. make clean).

Split-stream mutation testing on the other hand requires the external dependencies to be able to be reset to a specific state. While we build in support to reset local files to a specific state using a local git repository, more advanced dependencies like running, or even off-site, databases will need very specific commands and domain knowledge to enable the strategy to reset them to a specific state. The implementation for this will be different for each project.

These problems are not insurmountable, they are similar in nature to the problems that occur with distributed mutation testing.

Split-Stream Mutation Testing. Split-stream mutation testing allows us to reduce the execution overhead by an approximate factor of 2. However, this strategy requires too much knowledge of the system. It cannot easily be incorporated with external dependencies, e.g. databases. Our current recommendation is to not use the strategy.

Timed Out Mutants Overhead. For the bigger projects like CppCheck, we have seen that the execution time of the optimised mutation analysis consists mostly of timed-out mutants. Here, the timed-out mutants take up 93% of the reachable schemata mutation analysis. In our current approach, we detected the timeout once the total time for that mutant reached a threshold. For the CppCheck project, this means that a test that is stuck in an infinite loop would only be timed out after 45 seconds. As the CppCheck project has 3,745 test cases, the average test time is below 0.01 seconds. The impact of the timed-out mutants will be drastically reduced if we stop mutants not after a global threshold but after localised thresholds based on the individual tests.

D.7 Threats to Validity

As with all empirical research, we identify those factors that may jeopardise the validity of our results and the actions we took to reduce or alleviate the risk. Consistent with guidelines for empirical research (see [81, 82]), we organise them into four categories.

Construct validity: do we measure what was intended?
In essence, we want to know which parts of the mutation testing process are affected by building a mutant schemata and split-stream mutation optimisation on top of the Clang front-end, as we believed these strategies would eliminate the compilation and execution overhead of mutation testing. The mutant schemata strategy and extended reachable schemata strategy cause additional delays in some parts of the mutation testing process, but in return eliminate the need to compile every mutant individually. The real question is whether the time benefit from a single compilation outweighs the added delays. For this, we measured each part of an unoptimised mutation testing process and compared it to the optimisation strategies.

We implemented all of our mutants using the ternary (conditional) operator. This causes many additions to the code base, which impact the compilation time and test suite execution time. As mentioned before, this caused a large delay in the test suite execution time for the CheckCpp project as many ternary operators are written on frequently executed lines. In such cases, a different structure using switch-cases could have prevented such large delays.

While we chose a limited set of mutant operators (i.e. ROR, AOR, and LCR) and omitted mutants in *const* and templated expressions, we believe that our results represent the performance benefits one can achieve using the strategy. We believe so because the compilation times for the currently supported mutants are only impacted slightly, adding less than 25% to the compilation time for our biggest project. The test suite execution times were not impacted significantly except for the CheckCpp case where it caused an additional delay of 120%, which could be improved by using *switch* statements instead of *if* statements to select the mutants.

To ensure that the results are reliable and comparable, we used the same generate mutants method for all optimisation techniques. Additionally, we verified that the mutants we label as invalid mutants using the Clang front-end are the same mutants that actually caused compilation failures in the unoptimised baseline. This ensures that the same mutants are used across the various steps of the optimisation techniques.

Internal validity: are there unknown factors which might affect the outcome of the analyses?
We expected that the more mutants there are per line of code, the more the test suite execution time would be impacted as there is simply more code to execute. However, most of our projects did not show such results, they showed no difference in execution time with and without mutants.

As mentioned before, we believe that this might be due to the architecture of modern CPUs where out-of-order execution and pipeline depths/stalls can influence the effective performance. However, we did not investigate this sufficiently to draw firm conclusions.

We, however, did verify this in the case of CppCheck, for which the test execution times were heavily impacted. This was due to the number of mutants that can occur on a single line, drastically changing the execution time of that specific line. When that line is executed frequently, the performance of the entire system is affected considerably. On the other hand, the performance impact is minor if that single line is only executed once. Ideally, we need to investigate this further using a profiler.

To minimise environment factors, we locked the frequency of the CPU to the base frequency of the CPU and kept the computer in a well-ventilated space to prevent CPU throttling. While we used an older computer (Intel(R) Core(TM)2 Quad Q9650 CPU) architecture to run our experiments, we believe that this does not impact the analysis. The use of a hard drive instead of an SSD however will reduce the compilation times and likely lower the total speedups. We believe, however, that this should be limited, and that the benefits of the optimisation techniques would still be valid even when using an SSD.

We used the same infrastructure for the analysis of the five selected open-source projects. We used an Intel(R) Core(TM)2 Quad Q9650 CPU, with two 4GB (Samsung M378B5273DH0-CH9) DDR3 RAM modules (for a total of 8GB) and a 250GB Western Digital (WDC WD2500AAKX-7) hard drive. The PC was running Ubuntu 18.04.1 LTS (GNU/Linux 4.15.0-29-generic x86_64). Using an SSD would drastically influence the compilation times and negatively impact the total speedups. Using more RAM and/or a faster CPU might influence the compilation time and/or execution time, this, however, is presumed to be marginal.

External validity: to what extent is it possible to generalise the findings?
We evaluated our proof-of-concept tool on four open-source projects with different characteristics and one industrial project. These projects vary in size and in computational needs, varying from a low number of mutants per line of production code to high numbers. We believe that our results from these different projects represent the performance benefits one can achieve by using the different optimisation strategies.

Reliability: is the result dependent on the tools?
We took great care to ensure that external tools did not impact our timing results. The internal toolchain builds on very established components from the LLVM and Clang projects. We only measured the time it took to compile the project, generate the mutants, and execute them. We excluded any timings related to external tools or implementations like database access times to store the mutants. Our results thus represent the performance benefits of the optimisation strategies as implemented. On the other hand, we do see areas which can affect the performance. In some cases, where many mutants are listed on a single line, it might be faster to execute them using a *switch* statement than to use our *if* statement. In this case, the compiler can create a jump table (using consecutive indexes), i.e. an array of pointers, to directly jump to the correct label. Adding more mutation operators might also impact the performance.

D.8 Conclusion

In this paper we investigate to which extent the Clang front-end and its state-of-the-art program analysis facilities allow to implement existing strategies for mutation optimisation within the C language family. We present a proof-of-concept tool that allows us to collect detailed measurements for each of the mutation phases, i.e. generate mutants, compile mutants, and execute mutants. We validate the proof-of-concept tool on four open-source C++ libraries and one industrial component, covering a wide diversity in size, C++ language features used, compilation times, and test execution time. As such, we analyse the speedup from two perspectives (compilation time and execution time) assessing four optimisation strategies (exclude invalid mutants, mutant schemata, reachable mutant schemata, split-stream mutation testing). We address four research questions.

RQ1: How much speedup can we gain from mutant schemata? We could virtually eliminate the compilation overhead to the point that the compilation for all mutants was only slightly longer compared to the original compilation time, i.e. the time it takes to compile the project without any mutants. With the TinyXML2, Google Test and the industrial Saab project, the mutant schemata technique obtains a speedup between 4.87x and 5.16x. Yet, the obtained speedup depends on two factors: (a) the proportion of the project compilation time versus its execution time; (b) the number of mutants per line of production code. The latter is implementation specific and its impact can be reduced by changing the activation of the mutants from if statements to switch-cases. As a consequence of our implementation, for the other cases the speedup was less eminent: JSON has a speedup of 2.22x due to a high proportion of test suite execution time versus its compilation time. CppCheck obtained as speedup of 0.81x due to the increase in mutant execution time, attributed to the techniques implementation in combination with a high number of mutants/LOPC.

RQ2: How much speedup can we gain from split-stream mutation testing? In theory this strategy could yield a speedup of a factor 2, but in practice applying the strategy proves to be too demanding. Indeed, the split-stream mutation technique demands that dependencies need to be revertible to specific states of the execution. Reverting to the internal state demands too much specific knowledge about the design of the system under test, especially in the case data is stored in external databases and filesystems. Thus for the cases we investigated, we could not eliminate the execution overhead with the split-stream mutation testing strategy.

RQ3: How much speedup can we gain from eliminating invalid mutants? Compared to an *unoptimised* approach the reduction in the compilation overhead leads to a speedup by a factor between 1.07x and 1.12x. This is not all that much but it is a necessary prerequisite for the mutant schemata strategy. Since all mutants are injected simultaneously, the resulting program must compile without any errors. Invalid mutants are not acceptable, since they prevent the compilation (and the execution) of the mutated program. This is especially challenging for statically typed languages with many interacting features and unforgiving compilers (C, C++, ...). Since Clang has access to all of the project's type-information, we can programmatically ensure that any of the created mutants are statically correct.

RQ4: How much speedup can we gain from eliminating unreachable mutants? Compared to an *unoptimised* approach, excluding the completely unreachable mutants provides a speedup between 1.05x to 1.18x. One notable exception is the JSON project which had no unreachable mutants. This speedup stays approximately the same with the *mutant schemata* approach where the speedup is between 1.03x to 1.25x. This speedup, however, depends on the actual coverage of the test suite: a test suite with low coverage (thus reaching fewer mutants) yields a higher speedup. The *reachable schemata* technique goes one step further by not only excluding completely unreachable mutants but also excluding the test cases which do not reach the mutant on a mutant by mutant base. This gives an additional speedup between 1.83x and 12.98x over the schemata technique that excludes completely unreachable mutants. Providing a speedup between 2.29x and 12.76x over the normal schemata technique. Further speedups are possible by optimising the time-out function that e.g. detects mutants stuck in infinite loops. This can be achieved by specifying a time-out for each test case instead of a time-out for the global test suite.

Overall. In summary, we successfully demonstrated the feasibility of using the Clang compiler front-end for different optimisation strategies. With the reachable schemata strategy, we virtually eliminated the compilation overhead to the point that the compilation for all mutants was only slightly longer compared to the original compilation time and we reduced the execution overhead by only executing the test cases for individual mutants which actually reach said mutants. Compared to an *unoptimised* approach we achieve a maximum speedup of 23.45x and 30.52x on the JSON and Google Test projects with the *reachable schemata* strategy. Even for less ideal scenarios from the CPPCheck and TinyXML2 projects we achieve a speedup of 2.07x and 5.89x. These can be speedup further by two optimisations: Firstly, use *switch* statements for the mutant selection to reduce the technique overhead. Secondly, tailoring the time-out function, that e.g. detects mutants stuck in infinite loops, on a test by test basis instead of on the global test suite.

Finally, we report some lessons learned for deploying a mutant schemata tool using the Clang compiler framework. Most important is that we need a different, specialised approach for generating mutants in *const*, *constexpr*, *templates*, and *define* macro's. These statements are evaluated at compile-time, thus obstruct the runtime selection of the mutant required by the mutant schemata technique.

Acknowledgments

This work is supported by (a) the Research Foundation Flanders (FWO) under Grant number 1SA1519N; (a) the FWO-Vlaanderen and F.R.S.-FNRS via the Excellence of Science project 30446992 SECO-ASSIST.

Paper E

F-ASTMut Mutation Optimisation Techniques using the Clang Front-end

Sten Vercammen
University of Antwerp
Antwerp, Belgium

Serge Demeyer
University of Antwerp
Antwerp, Belgium

Markus Borg
RISE Research Institutes of Sweden
Lund, Sweden

Abstract

F-ASTMut is an open-source mutation testing research tool for the C language family based on manipulation of the abstract syntax tree. The tool is designed for detailed measurements, analysis, and tuning of optimisation techniques. The goal of F-ASTMut is to analyse the speedups of optimisation techniques to ultimately enable mutation testing in industrial settings. Currently, F-ASTMut features four optimisation techniques; an exclusion scheme for invalid mutants, a test-suit-scope reduction to only cover relevant mutants, mutant schemata, and split-stream mutation testing. The implementation relies on the Clang front-end, allowing future work to extend or build on top of our solution.

under revision in *Software Impacts 2023*

Keywords
mutation testing tool, clang, AST, C, C++

Code metadata

Nr.	Code metadata description	Please fill in this column
C1	Current code version	v0.2
C2	Permanent link to code/repository used for this code version	`https://github.com/Sten-Vercammen/F-ASTMut`
C3	Permanent link to Reproducible Capsule	`https://codeocean.com/capsule/3514968/tree/v1`
C4	Legal Code License	GPL 3.0
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	C++, Python, bash, SQL
C7	Compilation requirements, operating environments & dependencies	llvm, llvm-dev, clang, libclang-dev, cmake sqlite3, libsqlite3-dev, git, cmake, bash
C8	If available Link to developer documentation/manual	`https://github.com/Sten-Vercammen/F-ASTMut`
C9	Support email for questions	Sten.Vercammen@uantwerpen.be

E.1 F-ASTMut description

F-ASTMut is an open-source abstract syntax tree-based mutation testing research tool for the C language family. The tool implements a variety of optimisation techniques to speed up mutation testing. The main goal of mutation testing is to verify the fault detection capabilities of a test suite. For this, artificial faults —called mutants— are injected into the source code. For each fault, the test suite needs to be run. The fraction of mutants that result in at least one failing test case provides an indication of the fault detection capabilities of a test suite. The more mutants detected, the stronger the test suite.

As a research tool, F-ASTMut is built with the primary intent of enabling fine-grained performance analysis. The tool is built for collecting the impact statistics of various optimisation techniques. Each phase of the mutation testing process is separated, i.e. mutant generation, compilation, and execution [22]. For each of these phases, precise timing data is collected. Different optimisation techniques, or versions of said techniques, can then be compared against each other or analysed for further improvements.

E.2 F-ASTMut built-in techniques

F-ASTMut incorporates an unoptimised baseline technique and four complementary optimisations:

1. A detection technique to exclude invalid mutants.
2. A test suite reduction technique to execute unreachable mutants and an improved version which only executes those tests that reach that specific mutant.
3. A mutant schemata technique to reduce the compilation overhead by compiling all mutants at once.

4. A split-stream technique to reduce execution overhead by starting the mutant from the mutation point instead of from the beginning of the test suite.

The same method for mutant generation is used for all the optimisations, ensuring that the resulting speedup of each optimisation can be compared.

Figure E.1 illustrates the different optimisations and the impact they have on the compilation time and test execution time. The figure should be read from bottom to top, left to right.

- **Unoptimised**. First, (and this applies to all optimisations) the mutants themselves must be generated; shown in the yellow box at the bottom. This is achieved via a visitor walking over the abstract syntax tree provided by the Clang compiler front end. Then, for each injected mutant the complete source code must be recompiled. We divided this into three categories: 1) the mutants that are reachable, shown by the rather large orange box; 2) the unreachable mutants; and 3) the invalid mutants. We consider the last two as overhead as the unreachable mutants will never be detected during execution, their result is thus predetermined. In the unoptimised approach we represent this, and the other overhead, in blue. The invalid mutants will fail to compile, their result is thus also predetermined.

  At this point in time, we have a compiled version of the system for each injected mutant. What is left to do is run all the tests for each of the mutants, shown in the green and red boxes marked with T1–T5; green marks a passing test (mutant is not killed) while red marks a failed test (mutant killed). Ideally, each mutant is killed.

  The first optimisation included in F-ASTMut is within the generation of the mutants. Here, the semantic analyser of Clang allows us to detect invalid mutants and exclude them; this is included in the yellow box.

  For the second optimisation, code instrumentation is utilised to identify unreachable mutants; shown in the purple box.

  Utilising both optimisations allows for the exclusion off the blue boxes from the unoptimised approach.

- **Schemata**. The schemata technique injects all mutants at once, using a run-time configuration parameter to select the appropriate mutant. This technique requires that all injected mutants are valid, hence the requirement for the additional step at the beginning: “Exclude Invalid Mutants”. Here we only need to compile the system once, hence the orange block showing the compilation time is reduced and the overhead of compiling invalid mutants is also removed. The test execution time (the red and green boxes) remains unaffected.

- **Reachable Schemata**. An improved version of the reachable mutants is included in F-ASTMut. Here, instead of only detecting completely unreachable mutants, the technique detects which test cases reach and execute the mutant. We need an additional analysis (the purple block “Detect Reachable Mutants/Tests”) but now we save a lot during test execution (indicated by the reduced set of red and green blocks at the top).

- **Split-Stream**. The last optimisation technique starts the mutant from the mutation point instead of from the beginning of the test suite. Compared to the previous optimisation, we can now reduce the test execution time for each individual test.

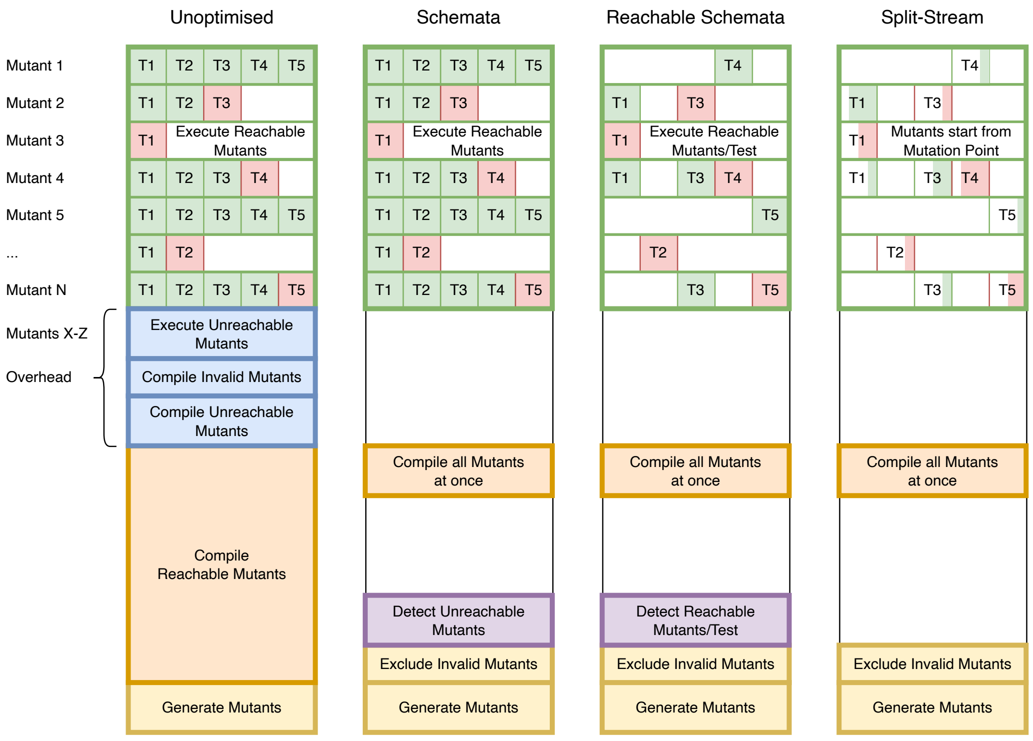

Unoptimised
Schemata
Reachable Schemata
Split-Stream
Mutant 1
Mutant 2
Mutant 3
Mutant 4
Mutant 5
...
Mutant N
Mutants X-Z
Overhead
T1
T2
T3
T4
T5
Execute Reachable Mutants
Execute Reachable Mutants/Test
Mutants start from Mutation Point
Execute Unreachable Mutants
Compile Invalid Mutants
Compile Unreachable Mutants
Compile Reachable Mutants
Compile all Mutants at once
Detect Unreachable Mutants
Detect Reachable Mutants/Test
Exclude Invalid Mutants
Generate Mutants

Figure E.1: F-ASTMut Implementation strategies with algorithm steps (first step at the bottom).

E.3 F-ASTMut underlying technology

The Clang project provides a language front-end and tooling infrastructure for programming languages in the C family […] for the LLVM project [https://clang.llvm.org]

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. […] capable of supporting both static and dynamic compilation of arbitrary programming languages. [https://LLVM.org]

LLVM is a collection of compilation tools designed around a low-level language-independent intermediate representation, the LLVM IR. The project includes frontends that translate source code to LLVM IR, optimisers that rewrite the LLVM IR to become faster, and backends that generate machine code from the LLVM IR for different architectures.

Clang is the most well-known front-end for LLVM. It supports languages in the C family, like C, C++, and Objective-C, among others. Internally, Clang represents programs as abstract syntax trees (ASTs). Clang also includes a semantic analyser that allows for type-checking and other compile-time checks. In addition, Clang contains several libraries based on the visitor pattern, allowing more analyses or transformations to be added to the front-end.

LLVM and Clang serve as the *de facto* standards for building static analysis tools for the C language family. These tools mutate the program either at the AST level or at the LLVM IR level. Doing the mutations at the LLVM IR level has the advantage that they will work for any frontend, but the disadvantage is that mutants injected in the LLVM IR are difficult or even impossible to trace back to a source representation in the original code under test, which allows for the generation of many invalid mutants [59].

The AST representation of a program, on the other hand, is close to the source code. Hence, mutating at this level provides good traceability, as these mutants can be represented in the source code. Another advantage of mutating at the AST level is that the front-end semantic analyser can be used to ensure that the mutated code is compile-time correct, effectively eliminating the acknowledged issue of invalid mutants.

F-ASTMut relies on the LibTooling library of Clang, and its ability to iterate through all declarations, statements, and expressions from the AST. For each of these declarations, statements, and expressions, variations will be created, i.e. the so-called mutants. This is done using mutation operators that describe which pieces of code shall be changed and according to which rules. For example, the AOR (Arithmetic Operator Replacement) mutation operator describes that an arithmetic operator shall be replaced with other arithmetic operators. A multiplication '$*$' operator residing in the binary expression '$a * b$' could, for example, create mutants by replacing the multiplication with the '$+$', '$-$', '$/$', and '%' operators.

To ensure traceability of the operator replacement, F-ASTMut stores the filename in which the mutant occurs, the offset to the beginning and the end of the original operator, and the mutant itself. For each executed mutant, F-ASTMut stores whether or not the mutant was reached, how long it took to run the corresponding test cases, and whether or not the mutant was killed. As some mutants might cause infinite loops during execution, the user can set a maximum execution time for each mutant. If the test suite is not able to finish executing within this time, F-ASTMut stops the test execution and records that the mutant *timed out*. Mutants that are timed out can be considered as killed, as they would also time out in a continuous integration test. This information can then be extracted to form a report to inform developers which mutants were never detected by the test suite, as well as where in the source code they reside.

E.4 Related Work

In Table E.1 we list the most prominent mutation testing tools that are based on Clang and/or the LLVM IR in alphabetical order with their features and optimisations. We briefly explain which optimisations they incorporated and refer to quantitive evidence if present.

AccMut (IR-based) features a mutant schemata approach and an optimisation called *modulo states* that applies the mutant schemata technique on local states. They have demonstrated an average speedup of 8.95x over a mutant schemata approach [60].

CCmutator (IR-based) is an LLVM IR mutation testing tool specifically designed to mutate concurrency constructs [94].

Dextool mutate (AST-based) is an open-source framework created for testing and static analysis of (often safety-critical) code[1]. It allows for distributed mutation testing in combination with a mutant schemata approach and can exclude mutants located in code that is not covered by the test suite.

Mart (IR-based) currently supports 18 different operator groups (with 68 *fragments* and 816 operators) [95]. These operator groups match against the LLVM IR syntax to create the mutants. Additional operator groups can be implemented by the user to further extend its capabilities.

[1]`https://github.com/joakim-brannstrom/dextool`

Table E.1: Clang and LLVM IR Mutation Testing Tools

Tool Name	**Mutation level**	**Mutation Operators**	**Mutation Optimisation**
AccMut	LLVM IR	AOR, ROR, LCR, SDL, ...	Mutant Schemata, modulo states
CCmutator	LLVM IR	Concurrency Mutation Operator	
Dextool mutate	AST	AOR, ROR, LCR, SDL, UOI	Distribution, Mutant Schemata, Exclude unreached mutants via code coverage
Mart	LLVM IR	Operator groups	Trivial Compiler Equivalence
MuCPP	AST	Class Level Mutants	Reduced Mutants Set
Mull	LLVM IR	LLVM fragments	Limit total mutants based on call-depth
SRCIROR	AST	AOR, LCR, ROR, ICR	Trivial Compiler Equivalence, Exclude unreached mutants via code coverage

Mart has an in-memory implementation of Trivial Compiler Equivalence to eliminate equivalent and duplicate mutants [63].

MuCPP (AST-based) generates mutants by traversing the Clang AST and storing the mutants in different branches using a version control system [96]. MuCPP implements mutations at the class level. These include mutations related to inheritance, polymorphism and dynamic binding, method overloading, exception handling, object and member replacement, and more. The study is aimed at reducing the total number of mutants that need to be executed by eliminating so-called unproductive mutants. These include equivalent mutants, invalid mutants, easy-to-kill mutants, and mutants in dead code.

Mull (IR-based) is an open-source mutation testing tool[2] which modifies fragments of the LLVM intermediate representation (LLVM IR). It only needs to recompile the modified fragments in order to execute the mutants, keeping the compilation overhead low [59]. Mull includes a *do-fewer* optimisation where you can limit which mutants are executed to only those mutants that are within a certain call-depth starting from the test case.

SRCIROR (AST or IR-based) is a toolset which can be set up to perform the mutation testing at the AST level or at the LLVM IRlevel [97]. Both variants implement the *AOR, LCR, ROR, ICR* mutation operators. SRCIROR allows filtering out unreachable mutants based on code coverage metrics. It also allows to filter out some equivalent mutants using trivial compiler equivalence [63].

> The current state-of-the-art demonstrates that mutant analysis for the C language family is possible on top of the LLVM and Clang compiler framework. However, the extent to which the various optimisation strategies allow reduced compilation and execution overheads is unknown. In particular, there exist no tool that allows for detailed measurements and analysis of different optimisations and/or their combinations.

[2]`https://github.com/mull-project/mull`

E.5 Impact of F-ASTMut

Using F-ASTMut, we have seen mutation testing speedups of up to 30.5x compared to the baseline, an unoptimised implementation [99]. F-ASTMut allows researchers to measure the speedup impact and potential delays of mutation testing optimisations. F-ASTMut also includes prominent mutation optimisation techniques such as mutant schemata, split-stream mutation testing and a test suite reduction technique which only executes those tests that reach that specific mutant. We made the tool open-source so that it can be extended and/or adapted. Researchers can utilise this tool to determine where the bottlenecks of mutation optimisation techniques lie, how to act upon them improve them and quantitatively analyse the improvements. For example, we have observed that our mutant schemata technique, which compiles all mutants at once and allows the activation of the mutants at runtime, causes a considerable execution overhead when many mutations exist in the same statement. F-ASTMut allowed us to determine the cause of the delay and how to improve the implementation of the optimisation. F-ASTMut also allows us to quantify any improvements we, or other researchers, make to optimisation techniques.

E.6 F-ASTMut research utilisation

The main part of the tool is implemented in the main.cpp file under the *tool* folder. This part is responsible for analysing the target source code, generating and exporting the mutants, and instrumenting the same source code with the injected mutants. Our tool also has accompanying scripts that drive the execution of the specific optimisations. Each optimisation is located in its own folder. Researchers can extend the main part of the tool or adapt the accompanying scripts to implement their own optimisations or run their own experiments.

The functionality of F-ASTMut can be manipulated by setting the different flags listed in Table E.2. Below we give a brief overview of the possibilities of F-ASTMut.

Table E.2: Optimisation flags F-ASTMut

Optimisation	flags
Traditional implementation	EXPORT_MUTANTS & EXPORT_NON_COMPILING_MUTANTS
Detecting reachable mutants	EXPORT_REACHABLE_MUTANTS
Mutant Schemata	SCHEMATA & EXPORT_MUTANTS
Split-Stream	SPLIT_STREAM

F-ASTMut can generate the mutant and store them into a numbered mutants.csv file. This file contains the start and end position of the original code, and the mutated code to take its place. As F-ASTMut relies on the Clang infrastructure it can automatically determine whether a generated mutant can be compiled or not. Therefore we have two different flags to export the valid and non-compilable mutants. Optimisation techniques can either directly plug in to the main tool or utilise the accompanying scripts to process the generated mutants.

For our *traditional implementation*, we developed an accompanying *traditional* script to individually insert the generated mutants into the code base, compile them separately and run the corresponding test suite.

Our EXPORT_REACHABLE_MUTANTS flag instruments the code base by putting markers inside the positions of all mutants. Running the accompanying *reachable* script executes the code base and collects which mutants are reachable by which test cases. This information can then be utilised to speed up the traditional implementation or another optimisation, e.g. mutant schemata.

For our *mutant schemata technique*, the SCHEMATA and EXPORT_MUTANTS flags need to be activated. The technique is implemented in the main tool and directly instruments the code base to include all mutants at once. The accompanying *schemata* script extracts the mutant numbers from the exported mutants.csv file. It then activates each mutant individually and runs the test suite against the mutated source-code. Alternatively, only the reachable mutants can be activated by combining the previous technique.

The split-stream optimisation is activated using the SPLIT_STREAM flag. Just like the schemata optimisation, it instruments the code base and relies on a script to run the mutants against the test suite.

E.7 F-ASTMut Limitations and Future Work

In version 0.2 of F-ASTMut, we have implemented the Relational Operator Replacement (ROR), Arithmetic Operator Replacement (AOR), and Logical Connector Replacement (LCR) (see Table E.3). The current version of F-ASTMut can be readily extended to support other binary mutation operators and unary operators. Other mutation operators, like the Access Modifiers Change (AMC) which can change private functions to public functions, might need more development effort.

Table E.3: Currently implemented mutation operators in F-ASTMut

Code	Short	Description
ROR	Relational Operator Replacement	Replace a single operator with another operator. The relational operators are <,<=,>,>=,==,! =
AOR	Arithmetic Operator Replacement	Replace a single arithmetic operator with another operator. The operators are: $+, -, *, /, \%$
LCR	Logical Connector Replacement	Replace a logical connector with the inverse. The logical connectors are: ||, &&, |, &

The driver for mutant schemata relies on information from outside the program to control the activation of the mutants by setting the MUTANT_NR variable. The MUTANT_NR variable is initialised at runtime and will thus never be const. This means that F-ASTMut cannot use the variable inside const and constexpr functions, as these functions are evaluated at compile time and the MUTANT_NR value cannot be known at compile time. Consequently, F-ASTMut cannot mutate const and constexpr functions in the same way as non-const functions. This includes type definitions (e.g. *using* ...), template arguments, static_asserts, etc. Hence, in version 0.2, F-ASTMut does not implement these kinds of mutations. This can be implemented by creating separate values and/or functions for each mutated operation and by selecting the correct one everywhere in the source code where they are used (e.g. const val becomes const val_0, const val_1, const val_2, ...).

The current implementation of the mutant schemata technique uses the ternary operator, which is the short-hand version of an if statement. This turned out to cause some execution delays when many mutants are located in a single expression. We envision that this can be optimised by implementing the mutant schemata technique using switch cases [99]. An improvement to the detection mechanism for mutants stuck in infinite loops should be addressed in future work. Currently, there is a single timeout based on the execution time of the entire test suite instead of individual test cases. As almost half of the analysis time is spent by these mutants, specifying a reduced timeout per test case would further speed up the analysis.

Paper F

Validation of Mutation Testing in the Safety Critical Industry through a Pilot Study

Sten Vercammen
University of Antwerp
Antwerp, Belgium

Markus Borg
Lund University
Lund, Sweden

Serge Demeyer
University of Antwerp
Antwerp, Belgium

Abstract

Mutation testing is the state-of-the-art technique to evaluate the fault-detection capabilities of a test suite, but its adoption has been limited. In this paper, we aim to investigate where mutation testing fits within the existing test strategies, whether the mutation analysis can be done in a timely manner, and which pain points remain to be tackled for industrial integration.

For this, we performed open format interviews with two companies developing safety critical software. Our first case had no experience with mutation testing, allowing us to analyse and aide them with the setup, integration, and the mutation analysis. Our second case has 5 years of experience with mutation testing, providing us with a mature view on mutation testing in practice.

Our study found that mutation testing can be combined with continuous integration and offload the work of the human reviewer by providing an initial code-quality review. Equivalent mutants appear to be a less prevalent obstacle, although flaky mutants are a concern that needs to be addressed. Overall, the industrial perception of mutation testing is evolving as more organisations recognise the potential benefits of the technique and work to address its limitations and challenges.

Submitted for peer review

F.1 Introduction

Today, mutation testing is acknowledged within academic circles as the most promising technique for assessing the *fault-detection capability* of a test suite [22, 23]. Furthermore, mutation testing has been shown to be superior to simple code coverage metrics in discovering test suite weaknesses [26]. Still, the industrial adoption of mutation testing is limited. One of the major problems of mutation testing is that is computationally expensive. However, recent advances have sped up the mutation testing analysis significantly. In this paper, we present an empirical study on the perception of mutation testing in industry. We explore the perceptions and attitudes of professionals in the field towards mutation testing.

Our goal is twofold. First, we investigate whether practitioners believe that mutation testing offers sufficient benefits for use in industrial contexts. Second, we study whether the recent mutation testing advances are sufficient to meet the computational performance requirements for industrial adoption. By collecting empirical data on the level of awareness and understanding of the technique, as well as the factors that influence its perceived value and utility, this study provides insights into the adoption and implementation of mutation testing in industry. Furthermore, we explore whether other significant pain points remain to be tackled for widespread industrial use.

The rest of the paper is structured as follows. In Section F.2, we elaborate on the concept of mutation testing and list related work. In Section F.3, we describe the design of our research methods, present the studied cases, and the data collection and analysis procedures. This naturally leads to Section F.4 where we discuss the results and lessons learned. As with any empirical research, our study is subject to various threats to validity and limitations which are listed in Section F.5. Finally, we draw conclusions in Section F.6.

F.2 Background and Related Work

Nowadays, it is common practice to execute automated tests on a continuous integration server to ensure that no changes to the project changed the behaviour of the project in an unexpected way. The tests themselves then serve as the quality gates of regression faults. Effective test suites maximise the likelihood of exposing faults [12]. Traditionally, code coverage is used to assess the strength of a test suite, revealing which statements are poorly tested. However, the code coverage technique has been shown to be a poor indicator of the test effectiveness of the test suite [17, 19]. Even the stronger coverage criteria, like full MC/DC coverage (Modified Condition/Decision Coverage, a coverage criterion often mandated by functional safety standards that target critical software systems, e.g., ISO 26262 and DO-178C) still do not guarantee the absence of faults [20, 21]. Hence, alternative techniques have been investigated. Today, mutation testing is acknowledged within academic circles as the most promising technique for assessing the *fault-detection capability* of a test suite [22, 23]. Mutation testing deliberately injects faults (called mutants) into the production code and counts how many of them are caught by the test suite. The more mutants the test suite can detect, the higher its fault-detection capability is – referred to as the *mutation coverage* or *mutation score*. An unoptimised mutation analysis would for each mutant, insert each mutant individually, compile the project and run the entire test suite against it [22]. This causes the mutation analysis to be computationally expensive, prohibiting it from being adopted in industrial settings. As a consequence, in the last decades, a lot of research has been devoted to optimise the mutation testing process [23, 52]. In this paper, we thus investigate whether the recent mutation testing advances are sufficient to meet the computational performance requirements for industrial adoption.

In 2017, Ramler et al. [34] performed an empirical study on the application of mutation testing for a safety-critical industrial software system. They applied a mutation analysis on a software system containing 60,000 LOC that generated more than 75,000 mutants. While they demonstrated the technical feasibility of applying mutation testing to the entire software system, they concluded that both the effort and costs required exceed what usually can be dedicated to unit testing. They also found that the computational execution time for the mutation analysis needed to be sped up drastically before it could be applied on a daily basis. Finally, they saw that on average, the time it takes to review a mutant and revise the tests took ca. 5 minutes. The large number of mutants inhibits a comprehensive use of mutation testing for the entire system.

F.3 Research Design

In this section, we present our research questions, the case companies, the data collection and the analysis procedures. The research design follows the guidelines for conducting and reporting case study research proposed by Runeson et al. [100].

We conduct a holistic multiple-case study in two development contexts represented by two different companies. For both contexts, the case under study, i.e. "the contemporary software engineering phenomenon in its real-life setting" [100], is mutation testing adoption.

F.3.1 Research Questions

The overall goal of this study is to explore the industrial perspective on mutation testing. This empirical study is the final step in a 5-year research project that focuses on speeding up the mutation analysis. In this work, we investigate how mutation testing fits with the current source code quality metrics, whether the mutation analysis can be done in a timely manner, and which pain points remain to be tackled. For this, we investigate the following research questions in the context of large organisations developing safety-critical software:

RQ1 How does mutation testing fit the existing test strategies?
RQ2 What are the practitioners' expectations of mutation testing before having any hands-on experience?
RQ3 What is the industrial perception of mutation testing utility after a pilot study?
RQ4 Which are the major obstacles for industrial mutation testing adoption?

F.3.2 Case Description

This section presents the two case companies under study. Both companies develop complex safety-critical software systems and are thus primary candidates for adopting mutation testing. Table F.1 presents an overview of the companies, the module where mutation testing was evaluated, and its corresponding contexts.

Company A is a large international company active in the power and automation sector. The department we study is part of a development organisation managing hundreds of engineers, with development sites in Sweden, India, Germany, and the United States. The development context is safety-critical embedded development in the domain of industrial control systems, governed by IEC 61511. Projects typically last 2–4 years and follow an iterative stage-gate project management model. The product is certified to Safety Integrity Level (SIL) 3 as defined by IEC 61508, mandating rigorous development and maintenance processes. The product's typical customers require safe process automation in very large industrial sites.

The components for which we performed mutation testing are part of a complex automation system. Parts of the system are very mature, with some legacy source code files that are more

Table F.1: Overview of the case companies.

	Company A	**Company B**
Domain	Process automation	Aerospace
Product	Automation system	Aeronautical product
Main standards	IEC 61508, IEC 61511	RTCA-DO178B/C
Developers	100-200	100+
Process model	Iterative with gate decisions	
	Case A	**Case B**
Project duration	2–4 years	>20 years, iterating 6-18 months
Modules size	multiple components totalling 34k LoC	100+ components ranging from 500 LOC to 100k LoC
Language	C/C++	C/C++
Line coverage	>80 %	100 %
Mutation Use	Pilot, Experimental setup, 1 month	Under evaluation by test improvement champions, integrated into development, 5 years
Mutation Optimisations	Compiler Integration	Compiler Integration, Mutant Filtering, Distribution, Trivial Compiler Equivalence (TCE)

than 30 years old. Most of the software is implemented in C/C++. Testing is a vital constituent in the safety assurance case, and code coverage metrics are carefully collected. The modules selected for mutation testing are modules belong to a new product currently in the early stages of development. The modules contain in total 34k lines of C/C++ code with a current line coverage goal, i.e., a target during active development, of >80%. Note that the selected modules do not conform to the safety standards, but were selected by the industry partner as particularly relevant.

Company B is a global company with offices and facilities in many countries around the world. The company has a long history of producing high-quality products for the aerospace sector and works with customers and partners in a variety of countries. The company has a decentralised structure, with business units focused on specific products and markets. It has a reputation for innovation and is often at the forefront of developing new technologies and solutions to meet the changing needs of its customers. The company develops safety-critical systems and adheres to 100% MC/DC testing (Modified Condition/Decision Coverage, the coverage criterion adopted for the highest Design Assurance Level (DAL) in accordance with the RTCA-DO178B/C standard).

The company has a team of test improvement champions with over ten years of mutation testing experience. They are continuously evaluating mutation testing within their organisation and have integrated it into the daily workflow for several teams. The components for which mutation testing is adopted vary from 500 LOC to 30k LOC and are part of a complex safety-critical system. The components are all written in C/C++ and range from simple libraries to complex applications.

F.3.3 Data Collection and Analysis Procedure

This study is based on data collected from mutation testing adoption in Company A (Case A) and Company B (Case B). The major differentiating factor between the two cases is the time frame during which they have been using or experimenting with mutation testing. On the one hand we have Case A, which represents a short-term mutation testing pilot for preliminary evaluation of the mutation testing technique and its potential benefits. We helped Company A to set up, execute and analyse the corresponding results on-site. The data collection spanned the entire time frame of the pilot and includes informal discussions and structured focus group meetings to gather broad perspectives from the practitioners. On the other hand we have Case B, which represents a long-term mutation testing pilot that started with a single team but now is constantly expanding within the organisation. The team we contacted had been actively involved in a five year research project on software testing improvements.

For both cases, we held semi-structured interviews guided by a list of open interview questions. We transcribed the interviews, summarised our findings, and verified the findings with the relevant stakeholders in the case companies to avoid potential misunderstandings.

The data analysis was done by one researcher and then verified by another. The analysis was carried out following a reviewed protocol and keeping a clear chain of evidence, i.e., allowing to trace the derivation of results and conclusions from the collected data [101].

Case A – Mutation testing adoption in Company A

The department under study in Company A had no experience with mutation testing before this study. We pitched mutation testing to them as a way to evaluate the fault-detection capabilities of their test suites and as one of the better indicators of code quality. As part of the study we would help integrate the mutation testing in their build server and perform an analysis of the mutation testing results. To do so, we then extended our tools to make them Windows-compatible as they utilise a Windows Azure build server. Finally, we scheduled a two-week time window in which we went to the company's premises to integrate our mutation testing tool with optimisations into their build server. On-site, we ran our optimised mutation testing techniques on the candidate projects, gathered the mutation testing results, and analysed their implications. We then presented the findings to the stakeholders of Case A in a focus group meeting with the first and second authors present. In this meeting, we discussed the results, how they should be interpreted and what would be needed to improve the mutation testing results. Most focus, however, was on the quality of the analysed test suite. We then elaborately discussed how to use and integrate our experimental mutation testing tool, but also other tools, into their build server in order for them to use the mutation score as an additional quality measure for their test suite.

We recorded the focus group meeting, allowing us to analyse it later, and to keep track of our data collection. From the transcription of the meeting, the first author summarised lessons learned and created a first version of the interview questions to aid subsequent discussions with both Company A and Company B. We validated the lessons learned in two steps. First, to mitigate researcher bias, the second author checked the summary and proposed areas for improvement. At the same time, the second author proposed refinements to the interview questions. Second, the summary was shared with the stakeholders of Case A to confirm the lessons learned. We then scheduled an additional interview to elaborate on our questions for which maturation had not already been reached. The final version of the interview questions with answer summaries is listed in Section F.3.3.

Case B – Mutation testing adoption in Company B

Since 2017 we have been in regular contact with the test improvement champion team of Company B regarding mutation testing and their optimisations. The team has been experimenting with and utilising mutation testing for over 5 years in their current project. Today the team has a fully integrated setup and workflow with mutation testing. The team agreed to share their findings regarding mutation testing with us in an interview. We sent our questionnaire by email, so that they could prepare their answers, and put safeguards in place to prevent the spread of confidential information. After their preparations, we scheduled and conducted the interview, aided by the questionnaire. The meeting was recorded and transcribed, allowing us to analyse it later, and to keep track of our data collection. From this transcription, the first author summarised the lessons learned. We validated the lessons learned in two steps. First, to mitigate researcher bias, the second author checked the summary and verified the findings. Second, the summary was shared with the stakeholders of Case B to confirm the lessons learned. The interview questions with answer summaries are listed in Section F.3.3.

Interview

Aided by the guideline questions, we asked the company representatives about their thoughts on mutation testing, their interests, whether or not they want to have mutation testing as an additional test quality metric, and whether or not mutation testing is sufficiently sped up for them to use in their development contexts. The interviews were conducted in an open format and were not limited to these questions. We summarised the lessons learned at the end of the meeting in order to prevent misunderstandings.

F.4 Results and Discussion

In this section, we answer the research questions based on the studied cases and summarise our findings. In order to do so, we first present a summary of the interview and the companies' answers in Table F.2.

Table F.2: Interview Questions

Company A	**Company B**
Which indicators do you use or consider good indicators of code quality in your context?	
We look at code quality as a broad term as it is more than just unit testing. We view requirement testing as a vital part of the code quality. For this, we keep a traceability matrix from the requirement specifications to the description of the test case/-functions. For the code itself, we have review processes, checklists and conventions in place and utilise static analysis tools. We run line- and statement coverage to verify our test effectiveness.	We utilise several tools and techniques to highlight potential problematic code areas. Our first and most obvious, but anecdotal, indicator of code quality is whether or not a reviewer complains about the understandability of the code. From our experience, we have seen that functions with high complexity are usually poorly tested and most likely contain bugs. We, therefore, have different design rules and guidelines in place to ensure the readability and understandability of the code. One example is to keep the lines of code in a single unit below 1,500 lines of code. Our static code analysis tools also highlight code with high McCabe complexity, as they might indicate problematic areas. As for test code, we have a guideline that, in general, a test case should not contain any *if* statements or other forms of branches.

Do you find this sufficient for your current needs, or do you see room for improvement?	
Today, our product is huge. We see that the areas for which we introduce late errors, or have pressing issues, mainly reside in sub-optimally tested parts of the code, which our current testing methods can amend.We do have some known "black-sheep" components due to our enforcement of global, average code coverage metrics.	This always comes down to a matter of cost and payoff. Tools should not get in the way of the user nor should they give them extra work. If the feedback is too complicated or overwhelming, we as users will stop using it. We did "recently" start to use more dynamic analysis tools such as address and undefined behaviour sanitisers. These instrument the binary to catch where and when undefined behaviour occurs.
Do you think mutation testing can help/helped with the previously mentioned "issues"?	
Something entirely different like mutant testing is probably a very good code quality indicator to have, but at the current time, it is difficult to qualify.	Mutation testing cannot find code with undefined behaviour. While we were aware of the issue, we needed to remove all undefined behaviours as otherwise, compiler optimisers will exploit these loopholes. What made this problem worse is that some mutants were causing undefined behaviour, and our testers were complaining about[1]. We thus started using the dynamic analysis tool, which allowed us to more quickly find the causes of the undefined behaviour. We could then utilise this information to stop generating the mutants that cause undefined behaviour.
What were your expectations of mutation testing? Which insights did you expect from mutation testing?	
We would expect a time-efficient tool, with easy and accessible information to analyse each test case individually, where it fails, why it fails and so on. Apart from running the tool in the continuous integration pipeline on pull request and nightly builds, we think that there is real value in running the tool locally to enable fast feedback. If we need to wait a day for feedback it takes forever to get rid of everything. Ideally, to prevent information overload, we would like to only see the feedback for the currently developed test case.	Before using mutation testing, we believed that mutation testing would automate the review of test cases and that it would replace code coverage tools. We came to the conclusion that mutation testing can never replace a human reviewing a test case. Mutation testing raises the bar and improves the overall quality of the test cases before a human reviewer looks at the test case. Such that the human can focus on understanding the test case not finding that you missed an assertion. Mutation testing will prompt you to correct this beforehand. Mutation testing covers the fundamentals of the test suite while the human can then focus on the traceability and how this relates to the requirements.

[1]To kill a mutant, additional tests are written. Then the mutated code can be run to see if the tests kill it. When the mutant is flaky, its execution path and results might differ from run to run, making it hard to kill.

Which insights did mutation testing bring?	
The mutation pilot showed us that we could improve our test suite. This led to an effort to increase the coverage with existing tools, as with our tight schedule, and the current stage of the newly introduced modules, we currently have no resources available to experiment with other tools.	In the beginning of the project we thought that we needed more mutants and that we needed to kill them all. But now we know that for our use case "less is more". We do not have enough time to kill them all. It is of no use if we generate mutants the user would not care about, like removing a log statement. We thus try to focus on mutants that provide us with the most value. For example, we no longer mutate the "for" statements like: *for (int i = 0; i < 10; ++i)* from < to *!=* and/or increments (*++*). We have rigorous test procedures in place where these faults would be found. We also stopped using the statement deletion operator on single statements, as they generated too many mutants. We delete bigger blocks of data like the body of *for* loops and *if* statements. With these mutants, it is very easy to convince the users to either delete the code or write a test case for it. Another insight we gained is that for a human it can be hard to see that a test case is verifying the input and not the output, like verifying the default value of the output. Mutation testing helps a lot in this regard. We can also trace how many mutants a test case kills, so when a test case kills 0 mutants, we know that we need to review it. We intent to improve the resolution to individual assert statements to allow the detection of assert statements that kill no mutants.
In your context, what potential do you think mutation testing has to improve code quality/alleviate bugs from slipping into production?	
We have seen that in our current setup, it is possible that a fault can creep into the production code. From what we can remember these occur in parts for which we did not have, or had insufficient, unit tests. Mutation testing might help in finding errors earlier.	The most important part for us is that mutation testing raises the bar of the code quality so that the human reviewer can focus on the non-obvious parts. As an example, we manually went over 1,000 review comments and found out that at least 20% of them would have been directly found with mutation testing. Although this being anecdotal evidence for one specific project, it provided us with indications that mutation testing automatically handles the low-hanging fruits, that would otherwise go over to the expensive manual review processes. We also had one instance where mutants survived or were killed when we did not expect them to behave as such. We traced the mutant, from the test case to our software verification cases and all the way to our software requirement where it turned out that the requirement itself was wrong and we needed to fix the requirement.

Do you think it is worth investing in mutation testing, both from a time to "implement" perspective as from a developer perspective?	
If we had the time, it would be great to invest in a mutation testing tool to investigate if, and how much, it can improve the code. Adding yet another tool also means adding additional maintenance for the tool. This should not outweigh the benefits of the tool.	This mainly comes down to cost, person hours are much more expensive than machine hours. For us, mutation testing is worth it. Mutation testing finds the low-hanging fruits. We have seen that mutation testing offloads the review process, it automatically reviews the test cases and can preempt simple review comments like missing assert statements. Mutation testing can also find issues that are hard and/or take a long time to detect by human reviewers like verifying default outputs.
How do/would developers react to the large output from mutation testing, i.e., the many examples of mutants that the current test cases do not cover?	
This depends on the developer. Some like to get feedback, but some can get annoyed when they get a lot of feedback. Our developers understand the need for quality and well-tested code. They understand that we need to have safeguards in place to verify the code quality, be it mutation testing or line coverage.	We worked a lot on our tooling to filter out mutations as we know that a user will not care about all mutants. By design, we also exclude a lot of mutants, like the mutants in the "for" loop statements. In our context, if you have a lot of mutants it means you're doing it wrong as you need to filter them. In addition to our filtering options, our tool also allows you to annotate the source code to prevent specific mutants from affecting the mutation score. We want our users to consider mutation testing as an experienced test engineer, *who* does not get tired. *He* (pre-)reviews your test cases so that the human reviewers can focus on the non-obvious parts. *He* removes so much of the manual review work.

Based on the mutation output, did you identify a need to write more test cases given the results of mutation testing? In general, how willing are developers to extend existing test cases and/or add new test cases for existing functionality in order to improve the mutation testing score?	
The mutation output caused an increased effort to improve the code coverage on a local basis instead of a global average. For this many additional tests were written. Our developers understand that this is for a good reason.	We can easily convince our developers that test cases need to be added or extended when we show them the mutants that delete code blocks. If there is a tool that tells you, “hey there might be something wrong, would you like to know?” most developers would always say yes, and then it is a judgment whether an action is required or not. The mutation testing score itself is not a goal, we use it more as a trend to see how the test suite has evolved over time, and to see if we need to do something about it or not? But it does trigger the gaming instinct, where you always want to improve and to be better than your neighbour. We do not use mutation testing for fast feedback, but we look at it a couple of times during the sprint to see where to focus our test efforts or if we need to improve the test suite. If needed, we act upon it or plan for it in the next sprint. We also do not write test cases just to kill a mutant, as we do not have the time or budget for this. We write test cases for the software verification cases that trace back to the requirements, which kill the mutants. We do not need to kill all mutants, as not all mutants are equally important. Sometimes a verification case does not exist, so we might need to write a verification case to kill the mutants. Sometimes there is no requirement, this implies that the code is either incorrect or not needed (causing the code to be deleted, which is the most common outcome) or the requirements are wrong.

Which obstacles did you encounter with mutation testing? e.g. integration, scalability, interpretability, execution time.	
We integrated the tool on a separate pipeline in our Azure DevOps server. The execution time of the mutation testing tool allowed us to run the analysis on the unit tests of our core components. But it is not fast enough to run daily for our entire project It does seem fast enough to run locally for test cases under current development.	It took a long time to get to the workflow we have today. It took us two years to integrate and improve the mutation testing so that it can run on our integration server. We do not execute the mutation testing on a pull request basis, but we run it every hour. Here we only execute mutants for modified or new functionality together with 10 % of the oldest mutants. Every 10h all mutants are re-evaluated. We can do this as we exclude a lot of mutants and focus on the “productive” ones. This ensures that the execution time stays relatively short. We believe that we can scale the mutation testing company-wide, with our current setup and optimisations like mutant schemata, running on the change code, filtering mutants, and the annotations to not generate mutants. We also put a lot of effort into the interpretability of our HTML report. This is super important, as otherwise, our developers will not use it if it is not easy to understand. An unexpected consequence of our rigorous filter of mutants and mutation operators is that we generate almost no equivalent mutants. After applying the trivial compiler equivalence (TCE) technique, we virtually have no equivalent mutants left.

F.4.1 Lessons Learned

In this section, we present the general lessons learned from the utilisation of mutation testing in industrial settings.

Requirement Based Testing. In both cases, a traceability matrix between the requirements and test cases/functions is maintained. As Case B has integrated mutation testing into their workflow, they extended the traceability matrix with the information from the mutants. This allows them to easily identify test cases that do not kill any mutants, providing them with an additional indicator for potentially problematic areas.

Shift Left Testing. Both cases have rigorous test procedures in place which can detect errors that are not caught by the unit tests. Any new tool that essentially has the same purpose, should be more cost-effective and/or bring more value. A tool like mutation testing promises a stronger unit test suite and consequently promises to catch more errors earlier. This is known as shift left testing, catching errors earlier is generally preferred and assumed to be more cost-effective.

Test Effort. With mutation testing, any survived mutant can seem like a weakness in the test suite. Hence, one might try to kill all mutants, but the test effort required to kill all mutants is tremendous. This test effort requirement is a major barrier for industrial adoption. However, in Case B we saw that they took a pragmatic approach to mutation testing where less is more. They realised that not all mutants are interesting and that testers do not want to kill the non-interesting mutants, like the removal of a log statement. By tailoring the mutation operators and excluding the non-interesting mutants they reduced the test effort required for mutation testing whilst keeping the relevant quality improvement for their test suites. This is in line with the recent trend to detect and exclude *unproductive* mutants from the mutation analysis [35, 102–105].

Silver Bullet. Mutation testing is not a silver bullet. It should not be treated as a replacement for existing tools like static analysis, code coverage, or other dynamic analysis tools. Mutation testing is a complement to these tools and should therefore also be used as such. This means that while mutation testing could take over the functionality of some existing tools, like code coverage, the existing tools often provide the relevant information in a more timely and cost-effective manner.

The team in Case B has anecdotal evidence that the mutation testing technique pays off for them. For once project, they analysed over 1,000 review comments and found that mutation testing could prevent at least 20% of these review comments. The automated reviews from mutation testing remove the low-hanging fruits and increase the quality of the code. This frees up the expensive human reviewers and allows them to reason on a higher level instead of the fundamentals such as missing assert statements.

Tool Chain. The developers in Case A would like a localised mutation testing tool in addition to the tool on the continuous integration server, with fast feedback in order to create optimised test cases without intermission. Such a tool can be created by only executing the mutants relevant to the test case. Such techniques are already in development [2, 3]. The developers in Case B integrated mutation testing on their build server with hourly runs. They do not use mutation testing for fast feedback but rather look at it a couple of times during a sprint to see where to focus their test efforts. They either act on it immediately or plan it during the next sprint.

The difference in approaches could be due to the maturity and experience of mutation testing within the studied cases or due to the nature of the company, their culture, and/or the development approaches.

Equivalent Mutants. Academic literature indicates that equivalent mutants are a widespread and inconvenient phenomenon that takes up many human work hours to detect and resolve [41, 62, 63]. However, the pragmatic approach that Case B utilises for mutation testing where they reduce the number of mutation operators and mutants by excluding the non-interesting ones has an unintentional side effect that they generate almost no equivalent mutants. For the few they do, they can label them as such in the code and avoid future generation of them. As a result, equivalent mutants pose no issues for them with minimal effort.

Flaky Mutants. Some mutants can cause undefined behaviour, where multiple executions of the same mutant can have different outcomes, allowing a mutant to be detected in one run and survived in another. We call this flaky mutants [9]. While we have studied these academically, Case B has seen their frequent existence in an industrial environment. They now utilise more dynamic analysis tools such as undefined behaviour sanitisers to quickly find the causes of the undefined behaviour and extend this to the mutation testing tool to stop generating the mutants that cause undefined behaviour.

F.4.2 Research Questions

RQ1: How does mutation testing fit the existing test strategies? Mutation testing is not a replacement for existing tools. Killing all mutants in large industrial projects is considered impractical and too time-consuming. The human review cannot be replaced by mutation testing. A reduced set of mutant operators and mutants can be used to offload the workload for the human reviewers so that they can reason on a higher level instead of the fundamentals such as missing assert statements.

The mutation testing results can be intertwined with existing procedures and techniques to offer additional value. One example is the integration of mutants into the traceability matrix used for requirement-based testing in Case B. This can visualise whether there are any problematic areas or test cases that do not detect any mutants and might not yet fulfil their role as verification cases.

RQ2: What are the practitioners' expectations of mutation testing before having any hands-on experience? Mutation testing evaluates the fault-detection capabilities of a test suite. The more mutants killed, the better the test suite becomes. However, the test effort to kill all mutants is tremendous and cannot be applied for large-scale applications [34, 35, 105]. This together with the up-front effort to integrate mutation testing into the workflow provides a major barrier to industrial adoption.

RQ3: What is the industrial perception of mutation testing utility after a pilot study? Mutation testing exposes the limitations of a test suite. This provides the fundamental motive to write additional tests. Mutation testing with optimisations is now fast enough so that it can be integrated into the continuous integration servers. However, the bottleneck and barrier for widespread addition is the sheer number of mutants that needs to be killed. There is simply not enough time to kill all mutants. In large industrial projects, a hybrid solution where only the most interesting mutants are executed, offloading the human reviewers, might offer a good middle ground. This finding is in line with the recent trend to minimise the mutation set for the mutation analysis [35, 102–105]. While this works for Case B, mature mutation testing tools that integrate this ideology, in order to break down the initial startup effort and continuous human effort cost, are needed before companies will be willing to integrate mutation testing in their workflows.

RQ4: Which are the major obstacles for industrial mutation testing adoption? As we have seen with the setup in Case B, where they utilise a reduced set of mutation operators and exclude the non-interesting mutants, they generate almost no equivalent mutants. They only rely on the trivial compiler equivalence detection technique to detect, as the name suggests, trivially equivalent mutants. For the few, remaining equivalent mutants, they can label them as such in the code and avoid generating them in the future. As a result, equivalent mutants pose no issues and can be handled with minimal effort.

However, they observed many mutants with non-deterministic behaviour. These mutants can be killed in one run, but survive in another one. To find the root cause of undefined behaviour, they now utilise dynamic analysis tools such as undefined behaviour sanitisers. This can then also be utilised to gain insights into the mutation testing analysis to label mutants as flaky when they cause undefined behaviour.

F.5 Threats to Validity

As with all empirical research, we identify those factors that may jeopardise 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 [81, 82]), we organise them into four categories.

Construct validity refers to how well the chosen research method has captured the concepts under study. There is a risk that academic researchers and industry practitioners use different terms when discussing advanced software testing due to their different frames of reference. Previous work indeed indicated that the language used by academics and professional testers is poorly aligned [106]. To mitigate misunderstandings related to mutation testing terminology, we relied on *prolonged involvement.* The interviewees in Case B, have worked actively with mutation testing for several years. Case A, on the other hand, represents mutation testing novices for which we provided basic training as part of the two weeks spent by the first author at their site. The two weeks ended with a focus group discussion, after which the first and second authors iteratively developed the interview guide (presented in Section F.4) for the concluding interviews. Terms such as mutation scores, mutation analysis, and equivalent mutants were known by the interviewees at that time.

In addition, interview sessions with researchers may threaten the interviewees and lead them to respond according to assumed expectations, i.e., *desirability bias.* To reduce this threat, we guaranteed the interviewees' anonymity, and no rewards were provided for their participation. We also explicitly told the interviewees about their rights to withdraw at any time without requiring an explanation. Finally, we tried to minimize the potential for response biases by asking open-ended questions and allowing participants to provide additional context or explanations as they saw fit.

Internal validity concerns whether unknown factors might have affected the outcome of the analysis. Our study is exploratory, thus the conclusions in this paper are not primarily about causal relations. On the other hand, the identification of challenges somewhat resembles identifying factors in causal relations. We mitigate the risks of identifying incorrect factors by interviewing multiple roles at both case companies.

Two obvious differences between Case A and Case B are 1) the mutation testing tool used and 2) the amount of time the tool was used. We do not consider this a threat as we refer to Case A and Case B as separate cases and provide two separate context descriptions. However, as we used open interviews, the interview sessions were not identical, and different follow-up questions were explored. This might have influenced the way detailed avenues of the interviews were saturated. The use of the interview guide mitigated this threat, as well as using *observer triangulation* during sessions. Finally, we hypothesize that some contextual variation points might have influenced the answers provided by the interviewees, e.g., the use of an in-house mutation testing tool in Case B, the Case B interviewees' mandates as test improvement champions, the timing just before summer for the pilot study in Case A, and the longer project duration in Case B.

External validity refers to what extent it is possible to generalise the findings. For qualitative studies such as ours, findings can only be extrapolated using *analytical generalisation*, i.e., transferring findings based on a theoretical analysis of contextual factors and relating them to other cases. We support such generalisations by carefully describing the industrial contexts of Case A and Case B. Our findings suggest that mutation testing is a promising technique that has a place within the future test strategies of high-assurance software development. We believe this finding holds for C/C++ development in other safety-critical contexts. Furthermore, we believe that mutation testing, including the optimisations discussed in this paper, is ready for evaluation also in mission-critical software development in other domains, e.g., banking and insurance. We recommend that future studies target other industry sectors, possibly using other programming languages, to increase the external validity of our findings.

Reliability relates to whether the same outcome would be expected with another set of researchers. Interpretation is a core component of qualitative research and enabling exact replications is not a goal. Still, we reduced the influence by single researchers, i.e., *mono-researcher bias*, by the joint development of the interview guide, observer triangulation, and internal validation of interpretations within the research team. Finally, we used *member checking*, i.e., the interviews had the chance to validate our results at different levels of interpretation – from the transcribed interviews to the final drafts of the paper.

F.6 Conclusion

Our empirical study found that mutation testing is a powerful technique for evaluating the fault-detection capability of a test suite.The recent advances in mutation testing techniques and tools have helped to significantly improve the speed and scalability of the technique, allowing it to be integrated into the continuous integration process.

We have seen that mutation testing contributes to a requirements-based testing process, sometimes even revealing flawed or poorly phrased requirements. Mutation testing also helps with a shift-left testing strategy, but due to time constraints it may not be practical to attempt to kill all mutants in large-scale applications. Therefore Case B sought a middle ground and developed their own tooling, where they limit the number of mutation operators and only execute those mutants that are the most relevant or interesting to the developers. Mature, publicly available, mutation testing tools that integrate such ideology are needed, in order to break down the initial startup effort and continuous human effort cost, before companies will be willing to integrate mutation testing into their workflows.

In this study, we have seen that equivalent mutants are less of an issue than considered in academic circles. Besides trivial compiler equivalence, there are other ways to ensure that they do not hinder the testing organisation, e.g., carefully selecting the mutation operators, filtering out non-interesting mutants, and storing and labelling any remaining equivalent mutants. Flaky mutants, however, are a real concern that needs to be addressed when considering mutation testing in the long run.

Furthermore, our study indicates that mutation testing is not a replacement for human code review, it is a useful tool for offloading the identification and correction of low-hanging fruit. Overall, our study suggests that the industrial perception of mutation testing is evolving as more organisations recognise the potential benefits of the technique and work to address its limitations and challenges. By leveraging recent advances and addressing key issues such as performance, relevant mutants, and flaky mutants, mutation testing has the potential to become a widely-used technique for improving software quality in industry.

Acknowledgments

This work is supported by the Research Foundation Flanders (FWO) under Grant number 1SA1519N; We also would like to thank the anonymous partners for their involvement in this study.

Bibliography

[1] Sten Vercammen, Serge Demeyer, Markus Borg, and Sigrid Eldh. Speeding up mutation testing via the cloud: lessons learned for further optimisations. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–9, Oulu Finland, October 2018. ACM. ISBN 9781450358231. doi: 10.1145/3239235.3240506. URL `https://dl.acm.org/doi/10.1145/3239235.3240506`.

[2] Sten Vercammen, Mohammad Ghafari, Serge Demeyer, and Markus Borg. Goal-oriented mutation testing with focal methods. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 23–30, Lake Buena Vista FL USA, November 2018. ACM. ISBN 9781450360531. doi: 10.1145/3278186.3278190. URL `https://dl.acm.org/doi/10.1145/3278186.3278190`.

[3] Sten Vercammen, Serge Demeyer, and Lars Van Roy. Focal methods for C/C++ via LLVM: steps towards faster mutation testing. In *Proceedings of the 20th Belgium-Netherlands Software Evolution Workshop, Virtual Event / 's-Hertogenbosch, The Netherlands, December 7-8, 2021*, volume 3071 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021.

[4] Sten Vercammen, Serge Demeyer, Markus Borg, Niklas Pettersson, and Görel Hedin. Mutation Testing Optimisations using the Clang Front-end. 2022. doi: 10.48550/ARXIV.2210.17215. URL `https://arxiv.org/abs/2210.17215`.

[5] Sten Vercammen, Serge Demeyer, Markus Borg, and Pettersson. F-ASTMut Mutation Optimisations Techniques using the Clang Front-end. 2023.

[6] Sten Vercammen, Serge Demeyer, Markus Borg, and Pettersson. Mutation Testing Requirements Elicitation in Industry. 2023.

[7] Serge Demeyer, Ali Parsai, Sten Vercammen, Brent van Bladel, and Mehrdad Abdi. Formal Verification of Developer Tests: A Research Agenda Inspired by Mutation Testing. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles*, volume 12477, pages 9–24. Springer International Publishing, Cham, 2020. ISBN 9783030614690 9783030614706. doi: 10.1007/978-3-030-61470-6_2. URL `http://link.springer.com/10.1007/978-3-030-61470-6_2`.

[8] Zhong Xi Lu, Sten Vercammen, and Serge Demeyer. Semi-automatic Test Case Expansion for Mutation Testing. In *2020 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, pages 1–7, London, ON, Canada, February 2020. IEEE. ISBN 9781728162713. doi: 10.1109/VST50071.2020.9051637. URL `https://ieeexplore.ieee.org/document/9051637/`.

[9] Sten Vercammen, Serge Demeyer, Markus Borg, and Robbe Claessens. Flaky Mutants; Another Concern for Mutation Testing. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 284–285, Porto de Galinhas, Brazil, April 2021. IEEE. ISBN 9781665444569. doi: 10.1109/ICSTW52544.2021.00054. URL `https://ieeexplore.ieee.org/document/9440140/`.

[10] SP Ng, Tafline Murnane, Karl Reed, D Grant, and TY Chen. A preliminary survey on software testing practices in Australia. In *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, pages 116–125, Piscataway, NJ, USA, 2004. IEEE Press. doi: 10.1109/ASWEC.2004.1290464.

[11] 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 01641212. doi: 10.

1016/j.jss.2012.12.051. URL https://linkinghub.elsevier.com/retrieve/pii/S0164121212003561.

[12] Glenford J Myers. *The Art of Software Testing*. New York: John Wiley and Sons, 1979.

[13] Bram Adams and Shane McIntosh. Modern Release Engineering in a Nutshell – Why Researchers Should Care. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 78–90, Suita, Osaka, Japan, March 2016. IEEE. ISBN 9781509018550. doi: 10.1109/SANER.2016.108. URL http://ieeexplore.ieee.org/document/7476775/.

[14] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming Google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 233–242, Buenos Aires, May 2017. IEEE. ISBN 9781538627174. doi: 10.1109/ICSE-SEIP.2017.16. URL http://ieeexplore.ieee.org/document/7965447/.

[15] Rob M. Everything you need to know about Tesla software updates, 2014. [on line] https://www.teslarati.com/everything-need-to-know-tesla-software-updates/ — last accessed In May 2018.

[16] Jon Jenkins. Velocity Culture, 2011. Keynote Address at the Velocity 2011 Conference.

[17] Xia Cai and Michael R. Lyu. The effect of code coverage on fault detection under different testing profiles. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, July 2005. ISSN 0163-5948. doi: 10.1145/1082983.1083288. URL https://dl.acm.org/doi/10.1145/1082983.1083288.

[18] Cyrille Artho, Armin Biere, and Shinichi Honiden. Enforcer – Efficient Failure Injection. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085, pages 412–427. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 9783540372158 9783540372165. doi: 10.1007/11813040_28. URL http://link.springer.com/10.1007/11813040_28.

[19] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, pages 435–445, Hyderabad India, May 2014. ACM. ISBN 9781450327565. doi: 10.1145/2568225.2568271. URL https://dl.acm.org/doi/10.1145/2568225.2568271.

[20] 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, 1939-3520. doi: 10.1109/TSE.2015.2421011. URL http://ieeexplore.ieee.org/document/7081779/.

[21] 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, March 2015. ISSN 0010-485X, 1436-5057. doi: 10.1007/s00607-014-0418-5. URL http://link.springer.com/10.1007/s00607-014-0418-5.

[22] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, September 2011. ISSN 0098-5589. doi: 10.1109/TSE.2010.62. URL http://ieeexplore.ieee.org/document/5487526/.

[23] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation Testing Advances: An Analysis and Survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019. ISBN 9780128151211. doi: 10.1016/bs.adcom.2018.03.015. URL `https://linkinghub.elsevier.com/retrieve/pii/S0065245818300305`.

[24] J.H. Andrews, L.C. Brand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 402–411, St. Louis, MO, USA, 2005. IEEe. doi: 10.1109/ICSE.2005.1553583. URL `http://ieeexplore.ieee.org/document/1553583/`.

[25] 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*, pages 654–665, Hong Kong China, November 2014. ACM. ISBN 9781450330565. doi: 10.1145/2635868.2635929. URL `https://dl.acm.org/doi/10.1145/2635868.2635929`.

[26] Ali Parsai. Mutation analysis: an industrial experience report. Master's thesis, Universiteit Antwerpen, 2015.

[27] Rakesh Rana, Miroslaw Staron, Christian Berger, Jörgen Hansson, Martin Nilsson, and Fredrik Törner. Improving Fault Injection in Automotive Model Based Development using Fault Bypass Modelling. In *2nd Workshop on Software-Based Methods for Robust Embedded Systems*, 09 2013.

[28] Jon del Olmo, Javier Poza, Fernando Garramiola, Txomin Nieva, and Leire Aldasoro. Model driven Hardware-in-the-Loop Fault analysis of railway traction systems. In *2017 IEEE International Workshop of Electronics, Control, Measurement, Signals and their Application to Mechatronics (ECMSM)*, pages 1–6, Donostia, San Sebastian, Spain, May 2017. IEEE. ISBN 9781509055821. doi: 10.1109/ECMSM.2017.7945878. URL `http://ieeexplore.ieee.org/document/7945878/`.

[29] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, December 2018. ISSN 2523-3246. doi: 10.1186/s42400-018-0002-y. URL `https://cybersecurity.springeropen.com/articles/10.1186/s42400-018-0002-y`.

[30] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the Art. *IEEE Transactions on Reliability*, 67(3):1199–1218, September 2018. ISSN 0018-9529, 1558-1721. doi: 10.1109/TR.2018.2834476. URL `https://ieeexplore.ieee.org/document/8371326/`.

[31] Mohamed Mussa, Samir Ouchani, Waseem Al Sammane, and Abdelwahab Hamou-Lhadj. A survey of model-driven testing techniques. In *2009 Ninth International Conference on Quality Software*, pages 167–172. IEEE, 2009. doi: 10.1109/QSIC.2009.30.

[32] Burak Uzun and Bedir Tekinerdogan. Model-driven architecture based testing: A systematic literature review. *Information and Software Technology*, 102:30–48, October 2018. ISSN 09505849. doi: 10.1016/j.infsof.2018.05.004. URL `https://linkinghub.elsevier.com/retrieve/pii/S0950584918300880`.

[33] Richard Baker and Ibrahim 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, 1939-3520. doi: 10.1109/TSE.2012.56. URL `http://ieeexplore.ieee.org/document/6298894/`.

[34] Rudolf Ramler, Thomas Wetzlmaier, and Claus Klammer. An empirical study on the application of mutation testing for a safety-critical industrial software system. In *Proceedings of the Symposium on Applied Computing*, pages 1401–1408, Marrakech Morocco, April 2017. ACM. ISBN 9781450344869. doi: 10.1145/3019612.3019830. URL https://dl.acm.org/doi/10.1145/3019612.3019830.

[35] 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 (ICSTW)*, pages 47–53, Vasteras, April 2018. IEEE. ISBN 9781538663523. doi: 10.1109/ICSTW.2018.00027. URL https://ieeexplore.ieee.org/document/8411730/.

[36] Cyrille Dupuydauby. Mutation testing: first steps with stryker-mutator .net, 2019. https://gist.github.com/dupdob/60fefe8495c8ebe8638ffcffc98a8703 — last accessed March 2022.

[37] Phillip A. Laplante and Joanna F. DeFranco. Software Engineering of Safety-Critical Systems: Themes From Practitioners. *IEEE Transactions on Reliability*, 66(3):825–836, September 2017. ISSN 0018-9529, 1558-1721. doi: 10.1109/TR.2017.2731953. URL http://ieeexplore.ieee.org/document/8006260/.

[38] Allen T Acree, Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Mutation Analysis. Technical report, GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE, 1979.

[39] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, April 1978. ISSN 0018-9162. doi: 10.1109/C-M.1978.218136. URL http://ieeexplore.ieee.org/document/1646911/.

[40] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark 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, Buenos Aires, May 2017. IEEE. ISBN 9781538638682. doi: 10.1109/ICSE.2017.61. URL http://ieeexplore.ieee.org/document/7985697/.

[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, January 2014. ISSN 0098-5589, 1939-3520. doi: 10.1109/TSE.2013.44. URL http://ieeexplore.ieee.org/document/6613487/.

[42] A. Jefferson Offutt and Roland H. Untch. Mutation 2000: Uniting the Orthogonal. In W. Eric Wong, editor, *Mutation Testing for the New Century*, pages 34–44. Springer US, Boston, MA, 2001. ISBN 9781441948885 9781475759396. doi: 10.1007/978-1-4757-5939-6_7. URL http://link.springer.com/10.1007/978-1-4757-5939-6_7.

[43] Wei Ma, Thierry Titcheu Chekam, Mike Papadakis, and Mark Harman. MuDelta: Delta-Oriented Mutation Testing at Commit Time. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 897–909, Madrid, ES, May 2021. IEEE. ISBN 9781665402965. doi: 10.1109/ICSE43902.2021.00086. URL https://ieeexplore.ieee.org/document/9402071/.

[44] R.A. DeMillo, E.W. Krauser, and A.P. Mathur. Compiler-integrated program mutation. In *[1991] Proceedings The Fifteenth Annual International Computer Software & Applica-*

tions Conference, pages 351–356, Tokyo, Japan, 1991. IEEE Comput. Soc. Press. ISBN 9780818621529. doi: 10.1109/CMPSAC.1991.170202. URL `http://ieeexplore.ieee.org/document/170202/`.

[45] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 235–245, Lugano Switzerland, July 2013. ACM. ISBN 9781450321594. doi: 10.1145/2483760.2483782. URL `https://dl.acm.org/doi/10.1145/2483760.2483782`.

[46] A Jefferson Offutt, Roy P Pargas, Scott V Fichter, and Prashant K Khambekar. Mutation testing of software using a MIMD computer. In *in 1992 International Conference on Parallel Processing*, pages II–257–266., Boca Raton, Florida, 1992. CRC Press.

[47] K. N. King and A. Jefferson Offutt. A fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, July 1991. ISSN 00380644, 1097024X. doi: 10.1002/spe.4380210704. URL `https://onlinelibrary.wiley.com/doi/10.1002/spe.4380210704`.

[48] Choi Byoungju and Aditya P Mathur. High-performance mutation testing. *Journal of Systems and Software*, 20(2):135–152, February 1993. ISSN 01641212. doi: 10.1016/0164-1212(93)90005-I. URL `https://linkinghub.elsevier.com/retrieve/pii/016412129390005I`.

[49] Christian N Zapf. MedusaMothra-A distributed interpreter for the Mothra mutation testing system. Master's thesis, Clemson University, 1993.

[50] Iman Saleh and Khaled Nagi. HadoopMutator: A Cloud-Based Mutation Testing Framework. In Ina Schaefer and Ioannis Stamelos, editors, *Software Reuse for Dynamic Systems in the Cloud and Beyond*, volume 8919, pages 172–187. Springer International Publishing, Cham, 2014. ISBN 9783319141299 9783319141305. doi: 10.1007/978-3-319-14130-5_13. URL `http://link.springer.com/10.1007/978-3-319-14130-5_13`.

[51] Pablo C. Cañizares, Mercedes G. Merayo, and Alberto Núñez. EMINENT: Embarrassingly Parallel Mutation Testing. *Procedia Computer Science*, 80:63–73, 2016. ISSN 18770509. doi: 10.1016/j.procs.2016.05.298. URL `https://linkinghub.elsevier.com/retrieve/pii/S1877050916306494`.

[52] Macario Polo Usaola and Pedro Reales Mateo. Mutation Testing Cost Reduction Techniques: A Survey. *IEEE Software*, 27(3):80–86, May 2010. ISSN 0740-7459. doi: 10.1109/MS.2010.79. URL `http://ieeexplore.ieee.org/document/5452149/`.

[53] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. Balancing trade-offs in test-suite reduction. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 246–256, Hong Kong China, November 2014. ACM. ISBN 9781450330565. doi: 10.1145/2635868.2635921. URL `https://dl.acm.org/doi/10.1145/2635868.2635921`.

[54] 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, Vasteras, April 2018. IEEE. ISBN 9781538650127. doi: 10.1109/ICST.2018.00016. URL `https://ieeexplore.ieee.org/document/8367036/`.

[55] Sebastien Bardin, Mickael Delahaye, Robin David, Nikolai Kosmatov, Mike Papadakis, Yves Le Traon, and Jean-Yves Marion. Sound and Quasi-Complete Detection of Infeasi-

ble Test Requirements. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, Graz, Austria, April 2015. IEEE. ISBN 9781479971251. doi: 10.1109/ICST.2015.7102607. URL `http://ieeexplore.ieee.org/document/7102607/`.

[56] Mike Papadakis and Nicos Malevris. Mutation based test case generation via a path selection strategy. *Information and Software Technology*, 54(9):915–932, September 2012. ISSN 09505849. doi: 10.1016/j.infsof.2012.02.004. URL `https://linkinghub.elsevier.com/retrieve/pii/S095058491200047X`.

[57] Dominik Holling, Sebastian Banescu, Marco Probst, Ana Petrovska, and Alexander Pretschner. Nequivack: Assessing Mutation Score Confidence. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 152–161, Chicago, IL, USA, April 2016. IEEE. ISBN 9781509036745. doi: 10.1109/ICSTW.2016.29. URL `http://ieeexplore.ieee.org/document/7528957/`.

[58] 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*, pages 449–452, Saarbrücken Germany, July 2016. ACM. ISBN 9781450343909. doi: 10.1145/2931037.2948707. URL `https://dl.acm.org/doi/10.1145/2931037.2948707`.

[59] Alex Denisov and Stanislav Pankevich. Mull It Over: Mutation Testing Based on LLVM. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 25–31, Vasteras, April 2018. IEEE. ISBN 9781538663523. doi: 10.1109/ICSTW.2018.00024. URL `https://ieeexplore.ieee.org/document/8411727/`.

[60] Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. Faster mutation analysis via equivalence modulo states. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 295–306, Santa Barbara CA USA, July 2017. ACM. ISBN 9781450350761. doi: 10.1145/3092703.3092714. URL `https://dl.acm.org/doi/10.1145/3092703.3092714`.

[61] Alan Hevner and Samir Chatterjee. Design Science Research in Information Systems. In *Design Research in Information Systems*, volume 22, pages 9–22. Springer US, Boston, MA, 2010. ISBN 9781441956521 9781441956538. doi: 10.1007/978-1-4419-5653-8_2. URL `http://link.springer.com/10.1007/978-1-4419-5653-8_2`.

[62] A. Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, September 1997. ISSN 0960-0833, 1099-1689. doi: 10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U. URL `https://onlinelibrary.wiley.com/doi/10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U`.

[63] 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*, pages 936–946, Florence, Italy, May 2015. IEEE. ISBN 9781479919345. doi: 10.1109/ICSE.2015.103. URL `http://ieeexplore.ieee.org/document/7194639/`.

[64] Francis J. Lacoste. *Killing the Gatekeeper: Introducing a Continuous Integration System.* IEEE, 2019. doi: 10.1109/AGILE.2009.35. URL `https://ieeexplore.ieee.org/document/5261054`.

[65] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653, Hong Kong China, November 2014. ACM. ISBN 9781450330565. doi: 10.1145/2635868.2635920. URL `https://dl.acm.org/doi/10.1145/2635868.2635920`.

[66] 6 tips for writing robust, maintainable unit tests. `https://blog.melski.net/tag/unit-tests/`. Accessed: 2020-08-14.

[67] Avoiding flaky tests. `https://testing.googleblog.com/2008/04/tott-avoiding-flakey-tests.html`. Accessed: 2020-08-14.

[68] Eradicating Non-Determinism in Tests. `https://martinfowler.com/articles/nonDeterminism.html`. Accessed: 2020-08-14.

[69] No more flaky tests on the Go team. `https://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team`. Accessed: 2020-08-14.

[70] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. An empirical study of bugs in test code. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 101–110, Bremen, Germany, September 2015. IEEE. ISBN 9781467375320. doi: 10.1109/ICSM.2015.7332456. URL `http://ieeexplore.ieee.org/document/7332456/`.

[71] August Shi, Jonathan Bell, and Darko Marinov. Mitigating the effects of flaky tests on mutation testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 112–122, Beijing China, July 2019. ACM. ISBN 9781450362245. doi: 10.1145/3293882.3330568. URL `https://dl.acm.org/doi/10.1145/3293882.3330568`.

[72] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 80–90, Chicago, IL, USA, April 2016. IEEE. ISBN 9781509018277. doi: 10.1109/ICST.2016.40. URL `http://ieeexplore.ieee.org/document/7515461/`.

[73] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. Finding bugs by isolating unit tests. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 496–499, Szeged Hungary, September 2011. ACM. ISBN 9781450304436. doi: 10.1145/2025113.2025202. URL `https://dl.acm.org/doi/10.1145/2025113.2025202`.

[74] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. DeFlaker: automatically detecting flaky tests. In *Proceedings of the 40th International Conference on Software Engineering*, pages 433–444, Gothenburg Sweden, May 2018. ACM. ISBN 9781450356381. doi: 10.1145/3180155.3180164. URL `https://dl.acm.org/doi/10.1145/3180155.3180164`.

[75] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 312–322, Xi'an, China, April 2019. IEEE. ISBN 9781728117362. doi: 10.1109/ICST.2019.00038. URL `https://ieeexplore.ieee.org/document/8730188/`.

[76] Flaky mutants dataset. `https://robbe-claessens.be/dataset-flaky-mutants.csv`. Accessed: 2020-08-14.

[77] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 72–82, Hyderabad India, May 2014. ACM. ISBN 9781450327565. doi: 10.1145/2568225.2568278. URL `https://dl.acm.org/doi/10.1145/2568225.2568278`.

[78] P.B. Kruchten. The 4+1 View Model of architecture. *IEEE Software*, 12(6):42–50, November 1995. ISSN 07407459. doi: 10.1109/52.469759. URL `http://ieeexplore.ieee.org/document/469759/`.

[79] 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*, volume 10522, pages 148–163. Springer International Publishing, Cham, 2017. ISBN 9783319689715 9783319689722. doi: 10.1007/978-3-319-68972-2_10. URL `https://link.springer.com/10.1007/978-3-319-68972-2_10`.

[80] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990. ISSN 0734-2071, 1557-7333. doi: 10.1145/78952.78953. URL `https://dl.acm.org/doi/10.1145/78952.78953`.

[81] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, April 2009. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-008-9102-8. URL `http://link.springer.com/10.1007/s10664-008-9102-8`.

[82] Robert K. Yin. *Case study research: design and methods*. Number v. 5 in Applied social research methods series. Sage Publications, Thousand Oaks, Calif, 3rd edition edition, 2003. ISBN 9780761925521 9780761925538.

[83] Mohammad Ghafari, Carlo Ghezzi, and Konstantin Rubinov. Automatically identifying focal methods under test in unit test cases. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 61–70, Bremen, Germany, September 2015. IEEE. ISBN 9781467375290. doi: 10.1109/SCAM.2015.7335402. URL `http://ieeexplore.ieee.org/document/7335402/`.

[84] Alexandre Perez, Rui Abreu, and Arie van Deursen. A Test-Suite Diagnosability Metric for Spectrum-Based Fault Localization Approaches. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 654–664, Buenos Aires, May 2017. IEEE. ISBN 9781538638682. doi: 10.1109/ICSE.2017.66. URL `http://ieeexplore.ieee.org/document/7985702/`.

[85] Arie Van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95, 2001.

[86] Mohammad Ghafari, Konstantin Rubinov, and Mohammad Mehdi Pourhashem K. Mining unit test cases to synthesize API usage examples. *Journal of Software: Evolution and Process*, 29(12):e1841, December 2017. ISSN 20477473. doi: 10.1002/smr.1841. URL `https://onlinelibrary.wiley.com/doi/10.1002/smr.1841`.

[87] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016. ISBN 9781316773123.

[88] Chris Arthur Lattner. *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[89] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Pearson Education, Limited, Sydney, 2015. ISBN 9780134049878. OCLC: 1338835528.

[90] Clang: a C language family frontend for LLVM. LLVM Developer Group. `https://clang.llvm.org` — last accessed March 2022.

[91] Ali Parsai and Serge Demeyer. Do Null-Type Mutation Operators Help Prevent Null-Type Faults? In Barbara Catania, Rastislav Královič, Jerzy Nawrocki, and Giovanni Pighizzini, editors, *SOFSEM 2019: Theory and Practice of Computer Science*, volume 11376, pages 419–434. Springer International Publishing, Cham, 2019. ISBN 9783030108007 9783030108014. doi: 10.1007/978-3-030-10801-4_33. URL `http://link.springer.com/10.1007/978-3-030-10801-4_33`.

[92] Ali Parsai, Serge Demeyer, and Seph 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*, volume 11146, pages 102–118. Springer International Publishing, Cham, 2018. ISBN 9783319999265 9783319999272. doi: 10.1007/978-3-319-99927-2_9. URL `https://link.springer.com/10.1007/978-3-319-99927-2_9`.

[93] Nan Li and Jeff Offutt. Test Oracle Strategies for Model-Based Testing. *IEEE Transactions on Software Engineering*, 43(4):372–395, April 2017. ISSN 0098-5589, 1939-3520. doi: 10.1109/TSE.2016.2597136. URL `http://ieeexplore.ieee.org/document/7529115/`.

[94] 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)*, pages 722–725. IEEE, 2013. doi: 10.1109/ASE.2013.6693142.

[95] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. Mart: a mutant generation tool for LLVM. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1080–1084, Tallinn Estonia, August 2019. ACM. ISBN 9781450355728. doi: 10.1145/3338906.3341180. URL `https://dl.acm.org/doi/10.1145/3338906.3341180`.

[96] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Francisco Palomo-Lozano, Antonio García-Domínguez, and Juan José Domínguez-Jiménez. Assessment of class mutation operators for $C++$ with the MuCPP mutation system. *Information and Software Technology*, 81:169–184, January 2017. ISSN 09505849. doi: 10.1016/j.infsof.2016.07.002. URL `https://linkinghub.elsevier.com/retrieve/pii/S0950584916301161`.

[97] Farah Hariri and August Shi. SRCIROR: a toolset for mutation testing of C source code and LLVM intermediate representation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 860–863, Montpellier France, September 2018. ACM. ISBN 9781450359375. doi: 10.1145/3238147.3240482. URL `https://dl.acm.org/doi/10.1145/3238147.3240482`.

[98] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using mutant schemata. *ACM SIGSOFT Software Engineering Notes*, 18(3):139–148, July 1993. ISSN 0163-5948. doi: 10.1145/174146.154265. URL `https://dl.acm.org/doi/10.1145/174146.154265`.

[99] Sten Vercammen, Serge Demeyer, Markus Borg, Niklas Pettersson, and Görel Hedin. Mutation Testing Optimisations using the Clang Front-end. *arXiv preprint arXiv:2210.17215*, 2022.

[100] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. *Case Study Research in Software Engineering: Guidelines and Examples.* Wiley, 1 edition, March 2012. ISBN 9781118104354 9781118181034. doi: 10.1002/9781118181034. URL `https://onlinelibrary.wiley.com/doi/book/10.1002/9781118181034`.

[101] Robert K. Yin. *Case study research: design and methods.* Sage Publications, Los Angeles, fifth edition edition, 2014. ISBN 9781452242569.

[102] Bob Kurtz, Paul Ammann, Marcio E. Delamaro, Jeff Offutt, and Lin Deng. Mutant Subsumption Graphs. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 176–185, OH, USA, March 2014. IEEE. ISBN 9781479957903. doi: 10.1109/ICSTW.2014.20. URL `http://ieeexplore.ieee.org/document/6825656/`.

[103] 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, 2014. doi: 10.1109/ICST.2014.13.

[104] Leonardo Fernandes, Márcio Ribeiro, Luiz Carvalho, Rohit Gheyi, Melina Mongiovi, André Santos, Ana Cavalcanti, Fabiano Ferrari, and José Carlos Maldonado. Avoiding useless mutants. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 187–198, Vancouver BC Canada, October 2017. ACM. ISBN 9781450355247. doi: 10.1145/3136040.3136053. URL `https://dl.acm.org/doi/10.1145/3136040.3136053`.

[105] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and Rene Just. Practical Mutation Testing at Scale: A view from Google. *IEEE Transactions on Software Engineering*, 48 (10):3900–3912, October 2022. ISSN 0098-5589, 1939-3520, 2326-3881. doi: 10.1109/TSE.2021.3107634. URL `https://ieeexplore.ieee.org/document/9524503/`.

[106] Vahid Garousi and Michael Felderer. Worlds Apart: Industrial and Academic Focus Areas in Software Testing. *IEEE Software*, 34(5):38–45, 2017. ISSN 0740-7459, 1937-4194. doi: 10.1109/MS.2017.3641116. URL `https://ieeexplore.ieee.org/document/8048625/`.