

DISSERTATION  
submitted  
to the  
Combined Faculty for the Natural Sciences and Mathematics  
of  
Heidelberg University, Germany  
for the degree of  
Doctor of Natural Sciences

Put forward by

Thomas Bach, M. Sc.  
born in Zweibrücken, Germany

Oral examination:



THOMAS BACH

# TESTING IN VERY LARGE SOFTWARE PROJECTS

Advisor: Prof. Dr. Artur Andrzejak

Advisor: Prof. Dr. Peter Bastian

Copyright © 2020 Thomas Bach  <https://orcid.org/0000-0002-9993-2814>

This work is licensed under a [Creative Commons](#)  
“[Attribution 4.0 International](#)” license.



# Abstract

Testing is an important activity for software projects. However, testing requires effort and therefore creates so-called test costs. They consist of factors such as human effort for testing, or costs for required software and hardware. Test costs can contribute considerably to the total cost of a software project. Therefore, it is desirable to reduce test costs without affecting the quality of the software. With agile development processes and short test cycles, this aim is even more important.

Consequently, prior work propose a wide range of techniques for test cost reduction. However, while studying a very large industrial software project, we found that techniques proposed in related work can often not be applied for a project of such a size. We attribute this to problems that are only significant in large projects and may not be noticed in smaller projects.

The amount of test execution grows superlinearly over time in large projects. This superlinear increase cannot be sustainably solved by increases in hardware. Additionally, tests with long execution times have several negative consequences. A large number of changes and the long execution times of tests create a conflict that does not exist in smaller projects.

Therefore, approaches for test cost reduction must consider the specific characteristics of large projects. In our work, we tackle this challenge in multiple ways. We describe an approach to replace the superlinear increase in test executions with a linear increase by utilizing a multi-stage test strategy that provides economic incentives to decrease test costs. We also design an approach that extracts unit tests from system tests and utilizes the multi-stage test strategy to reduce test costs without reductions in quality. Our approach builds upon time-travel debugging and reverse-executes a system test to identify dependencies. We combine coverage-based differential analysis and source code analysis via a compiler plugin to enable accurate identification of the test core. Our approach extends previous work by creating maintainable source code instead of binary data and by focusing on the test core instead of extracting tests for all parts of the software.

In addition to our core contributions, we also provide two empirical studies that investigate the relationship between coverage data and faults. The first study investigates the distribution of bugs and coverage. The second study investigates Granger-causality between coverage and bugs.

Finally, as a byproduct of our core contributions, we provide approaches for automatically determining the creation of objects with complex dependencies, for automated mock recommendation, for combining combinatorial testing with coverage-based test amplification, for reductions of coverage data sizes, and we solve coverage-based algorithmic problems.

# Zusammenfassung

Das Testen von Software ist ein wichtiger Bestandteil der Softwareentwicklung. Es erzeugt allerdings auch sogenannte Testkosten. Diese können Softwareprojekte erheblich teurer machen. Insbesondere in Zeiten von agiler Softwareentwicklung sind Testzyklen kürzer geworden und die Testkosten entsprechend gestiegen. Deshalb ist eine Testkostenreduktion wichtig.

Bei der Analyse eines sehr großen Softwareprojekts haben wir festgestellt, dass viele Ansätze aus der Literatur nicht für ein Projekt dieser Größe übertragbar sind. Die Größe des Projekts führt zu besonderen Herausforderungen, die in kleineren Projekten nicht auftreten oder ignoriert werden.

In großen Projekten wächst die Anzahl der Testausführungen superlinear über die Zeit. Dies ist nicht nachhaltig mit Investitionen in Hardware ausgleichbar. Weiterhin gibt es mehr Tests mit langen Ausführungszeiten. Die große Anzahl von Änderungen und die langen Testlaufzeiten in großen Projekten erzeugen einen Konflikt, der in kleinen Projekten nicht auftritt.

Ansätze für die Testkostenreduzierung müssen deshalb die Gegebenheiten von großen Projekten berücksichtigen. Wir erweitern den Stand der Forschung um mehrere solche Ansätze. Zur Evaluation nutzen wir große C++ Projekte, ein industrielles und mehrere Open-Source Projekte.

Wir stellen eine Methode vor zur Überführung des superlinearen Wachstums von Testausführungen in lineares Wachstum mittels einer mehrstufigen Teststrategie mit ökonomischen Anreizen für die Testkostenreduzierung.

Weiterhin konzipieren wir eine Technik welche unit tests von system tests extrahiert. Diese Technik kombinieren wir mit der mehrstufigen Teststrategie um eine Testkostenreduzierung ohne Qualitätsverluste zu ermöglichen. Unsere Technik nutzt time-travel debugging. Dabei werden system tests rückwärts ausgeführt um Abhängigkeiten zu identifizieren. Wir kombinieren Differentialanalyse auf Coveragedaten und Quelltextanalyse mithilfe eines Compilerplugins zur akkuraten Bestimmung des Testkerns. Im Vergleich zu vorherigen Arbeiten erzeugt unsere Technik unit tests mit wartbarem Quelltext statt Binärdaten und wir fokussieren uns bei der Extraktion auf den Testkern anstatt für die gesamte Testausführung unit tests zu erstellen.

Weiterhin präsentieren wir die Ergebnisse von zwei empirischen Studien. Die erste Studie untersucht die Verteilung von Bugs und Coverage. Die zweite Studie untersucht Granger-Kausalität zwischen Coverage und Bugs.

Schließlich stellen wir noch Methoden vor, um Objekte mit komplexen Abhängigkeitsgraphen zu erstellen, um Vorschläge für mocks von Objekten zu erstellen, zur Reduzierung der Größe von Coveragedaten, und wir lösen algorithmische Probleme im Zusammenhang mit Coveragedaten.

# Acknowledgements

First and foremost, I am grateful for all the discussions, advice, and guidance provided by my advisor, Artur Andrzejak. He pointed me into interesting directions for my research and fostered collaborations with other researchers. He provided support in all possible cases and I hope I can learn from his patience and kindness. Will I ever create a paragraph of writing where he does not immediately have an idea of how to improve it further? Maybe I created such a hidden gem somewhere in this work.

I am thankful for all the many persons that contributed to this work and on my way to this thesis. This thesis would barely exist without them.

Neuenheimer Feld, the place where I met many friendly colleagues that supported me and my work. Thank you Diego Costa (How many plants survived in our office?), Lutz Büch (Do you see the optical illusion?), Mohammadreza Ghanavati (Awesome time management), Tuyen Le (Take care of our office!), Zhen Dong, Kai Chen, and several members of other groups.

I thank all members of the research group at our industry partner, who welcomed me and my research outside of their database area. Lucas Lerch (prospective Mitbürger), Frank Tetzl (How long would a chess game with you really take?), Robin Rehrmann (Can we merge good and bad niveau?), Stefan Noll (Your next significant contributions?), Tiemo Bang (No more french beer), Michael Brendle (How many shades are there between hot and cold?), Jonas Dann, Mehdi Moghaddamfar, former members who always shared their experiences, Georgios Psaropoulos, Ismail Oukid, Matthias Hauck, Florian Wolf, Michael Rudolf, Robert Brunel, Marcus Paradies, David Kernert, Elena Vasilyeva, and Arne Schwarz, he opened a lot of doors as group manager, even if I constantly annoyed him by persistent inquiries.

Several parts of this work would maybe not exist without the support of many persons at our industry partner who helped to cope with all complex details of a large software project. Thank you Ralf Pannemans, Johannes Haeussler, Jörg Wiemers, Sascha Schwedes, Janos Seboek, Magnus Bie-neck, Katherina Nizenkov, Arne Gerner, Timo Hochberger, Sascha Bastke, Christoph Haefner, Colin Joy, Katharina Schell, Andreas Bader, Robin Joy.

Highly beneficial for this work were also several collaborations with other researchers. Thank you Pavneet Kochhar, Artha Prana, David Lo, Richard Kuhn, and many others for the productive exchange.

Everyone supporting me in my private life. My significant other, Judith, who magically endured me all this time. And, by logic deduction, this work would not exist without my parents and recursively all their parents.

Despite best efforts, I may have inadvertently omitted someone. Please accept my apologies and be assured that I am grateful for your support.





# Contents

	Page
<b>1 Introduction</b>	<b>1</b>
1.1 Outline and Contributions . . . . .	3
<b>2 Background</b>	<b>7</b>
2.1 Software Quality Assurance and Testing . . . . .	7
2.2 Study Subject: A Very Large Software Project . . . . .	15
2.3 Summary . . . . .	21
<b>3 Code Coverage: Measure Test Execution</b>	<b>23</b>
3.1 Definitions . . . . .	23
3.2 Implementation Details . . . . .	28
3.3 Problems and Algorithms on Coverage Data . . . . .	31
3.4 Summary . . . . .	52
<b>4 On the Relationship Between Coverage and Faults</b>	<b>53</b>
4.1 Discussion . . . . .	53
4.2 The Impact of Coverage on Bug Density . . . . .	58
4.3 Granger-Causality between Coverage and Faults . . . . .	69
4.4 Combinatorial Testing . . . . .	94
4.5 Conclusions . . . . .	104
<b>5 Analysis of Approaches for Test Cost Reduction</b>	<b>105</b>
5.1 Background . . . . .	105
5.2 An Economic Approach for Test Cost Reduction . . . . .	109
5.3 Test Case Selection and Prioritization . . . . .	119
5.4 Size of Coverage Data . . . . .	122
5.5 Shared Coverage: Test Core Identification . . . . .	126
5.6 Nondeterminism in Testing . . . . .	136
5.7 Threats to Validity . . . . .	139
5.8 Conclusions . . . . .	140
<b>6 Dynamic Unit Test Extraction</b>	<b>141</b>
6.1 Dynamic Unit Test Extraction via Time-Travel Debugging . . . . .	142
6.2 Object Creation . . . . .	160
6.3 Mock Proposal . . . . .	181
6.4 Summary . . . . .	190
<b>7 Conclusions</b>	<b>191</b>
<b>8 Bibliography</b>	<b>193</b>

# List of Notations

$\{i_1, \dots, i_n\}$	A set of $n$ distinct items.
$\langle a_1, \dots, a_n \rangle$	A sequence, i.e., a list of $n$ ordered items with possible repetitions.
$\mathbb{N}^+$	The set of natural numbers without zero: $\{1, 2, 3, 4, \dots\}$ .
$\mathbb{N}_0$	The set of natural numbers with zero: $\{0, 1, 2, 3, 4, \dots\}$ .
Boolean	A data type that is restricted to the set $\{0, 1\}$ .
$\mathbb{B}^n$	A $n$ -dimensional vector space of $\mathbb{B} = \{0, 1\}$ .
$S_{\text{instr}}$	A sequence of instructions.
$S_{\text{code}}$	Source code, a sequence of instructions in a programming language.
$S_{\text{IC}}$	A sequence of items providing information about the execution of $S_{\text{instr}}$ .
$\{c_1, \dots, c_n\}$	A set of $n$ coverage files.
$A \times B$	The Cartesian product between two sets $A$ and $B$ : $\{(a, b) \mid a \in A \text{ and } b \in B\}$ .
$a \sim b$	A binary relation, i.e., $(a, b) \in U$ for a given $U \subseteq A \times B$ , where $a \in A, b \in B$ .
$[a]$	The equivalence class of $a$ : $\{b \in X \mid a \sim b\}$ for a given set $X$ and a relation $\sim$ .
$\mathbb{W}(c_i) = w_i$	A weight function $\mathbb{W}$ that associated a weight $c_i$ to an item $C_i$ .
$\mathbb{P}(S)$	The power set of the set $S$ : $\{U \mid U \subseteq S\}$ .
$\ln(x)$	The natural logarithm of $x$ , i.e., base $e$ logarithm of $x$
$H(n)$	The $n$ -th harmonic number: $\sum_{i=1}^n 1/i \leq \ln(n) + 1$ .
$O(f)$	Asymptotic behavior of function $f$ : $\{g(x) \mid \exists c, x_o > 0 : 0 \leq g(x) \leq cf(x) \forall x \geq x_o\}$ .

# 1 | Introduction

Computer software is a general term for a set of instructions that are interpreted and executed by computer hardware. In contrast to fixed hardware, software is flexible and can be modified without much effort. This allows for rapid changes and advances in software which are largely decoupled from the development of the underlying hardware. Given these capabilities, the development of software is nowadays an important task and the systematic approach for creating software is called software engineering<sup>1</sup> [233].

The flexibility allowed by software does not only provide benefits, but also comes with potential disadvantages. As software is executed by hardware, it can remain unclear what exactly would be the result of such an execution. This is in contrast to classical engineering fields, where a physical product can be viewed, measured or investigated more directly. In addition, software must follow a precise interface defined by the hardware, where a classical engineering product may be used in a more lenient scenario in practical usage. These disadvantages can make the validation<sup>2</sup> and verification of software<sup>3</sup> more complex compared to the same tasks for physical engineering products. Moreover, humankind has more experience with classical engineering of structures, tools, and machines. Today's classical engineering products are the result of centuries of experience and improvements whereas the engineering of software has only seen several decades.

These differences finally lead to the question of how can we ensure that software “works as expected”, i.e., *how can we verify and validate software products?* The answer is two-fold. First, we must define the *expectations*, and second, we must verify the *works as*. The first part leads to the field of requirements engineering, i.e., the process of analyzing, defining, documenting and maintaining requirements [233]. However, we do not focus on requirements engineering in this work and refer to the literature for further information [73]. The second part leads to the field of software quality and software testing. Quality refers to “the degree to which a set of inherent characteristics fulfills requirements” [136] and targets the question of what are appropriate quality requirements and how to fulfill them. Consequently, software quality refers to the “degree to which a software product satisfies stated and implied needs when used under specified conditions” [133]. Software testing refers to any measure that is used to gain information about the quality of software. In an iterative process, software testing can result in software quality improvements if the information gained by testing is consequently used to improve the software and the result is measured again regarding the defined standards.

<sup>1</sup> ISO/IEC 2382: “systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software” [135]. ISO/IEC 19759: “the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software” [233].

<sup>2</sup> “building the right product”

<sup>3</sup> “building the product right”

How can we verify and validate software products?

Quality – “the degree to which a set of inherent characteristics fulfills requirements” [136].

Software quality and testing are important for software development. Software that does not fulfill expected requirements can reduce the willingness of a user to use the software or to pay for the development and usage. A typical scenario is a crash, i.e., the software stops functioning properly and exits. Other scenarios are incorrect modifications of data or wrong calculation results. We call such an undesired effect a *failure* and we use the term *fault* to describe the cause of a failure. We may use the term *defect* or *bug* to refer to either a fault or a failure [233]. Software testing and quality assurance aim to detect and remove all such defects.

Fault, failure, defect, bug.

However, it might be hard or even impossible to detect and remove all defects [124]. Even software that is developed with rigorous software quality standards, such as software used for space missions, contains defects that result in catastrophic consequences<sup>4</sup> [78, 204]. There are multiple reasons why it might not be possible to create software without defects:

<sup>4</sup>Huckle and Neckel survey a wide range of software failures [129].

1. For any non-trivial software, it might not be possible to conclusively and precisely define the correct set of requirements. For instance, several million users of a software for word processing might not share the same understanding of its expected behavior, even if the functionality is well defined by its documentation.
2. Software interacts with hardware based on an interface and therefore has only limited control. For instance, caching or parallel processing might affect the execution of a program in unexpected ways.
3. Hardware can fail. In fact, all hardware parts provide only a statistical guarantee for correctness. For instance, DRAM errors can alter data.
4. It might not be worth the effort to create a software without defects considering the potentially low impact of some defects. For instance, a user might not complain if software created for a programming exercise crashes for a specific instance of inputs.

Can we find all defects in a software?

In addition to the theoretical impossibility to detect and remove all defects, there are also practical considerations that limit the effort for software quality and testing. Assuming that software is sold for income, the development costs of the software are important [242]. For a sustainable business, the costs to develop the software must be smaller compared to the total income which is, simplified, the retail price (*price*) multiplied with the number of times it was bought (*sales*). However, the quality of the software can influence sales. This leads to the challenging problem of finding the optimal effort for software quality. Increasing the effort for quality increases costs that may not translate in proportional sales. Reducing the effort for quality and therefore reducing the costs can result in sales that do not provide enough income to make the development sustainable. The exact optimal solution remains unclear due to uncertainties in measurements and predictions. For instance, it remains unclear how to predict the effect of an unknown failure on sales. Additionally, different usage scenarios might require different quality standards. Cases where defects can have severe results, such as loss of human lives, have different requirements compared to cases where a defect is barely noticeable, such as a wrong placement of a graphic in a video game. We conclude that software quality causes costs and these costs limit the maximum effort invested in quality.

Optimal effort for software quality?

Given a fixed bound of maximum costs, it is important to optimize testing, i.e., to reduce test costs while remaining the same degree of software quality. Such reductions on test costs can have significant effects. Google reports 150 million test executions per day in 2017 [191], Facebook reports 10 billion test executions per day in 2019 [184], Microsoft reports 100 000 test executions per day for some products in 2015 [116], or SAP reports 1 million test executions per day in 2017 [20]. Given these numbers, even a 10% reduction in test executions would result in a large amount of absolute cost savings. Furthermore, even for smaller projects, such an improvement can be noticeable if they improve the development workflow by reducing waiting times for developers.

Large test costs provide large opportunities for cost saving.

In addition to the general economic argument for test cost reduction, time spent on testing also affects the productivity of developers. Typically, a developer executes tests to verify changes and new functionality. Even if the test execution is automated, the test execution itself requires time, the test execution time. For this time, a developer can switch tasks or wait for the result of the test execution. In simple scenarios with execution times for tests below a second, the waiting time is negligible. However, in large projects, the waiting time can require several minutes to several hours. Assuming a test execution time of two hours, this has multiple effects:

1. Developers cannot re-run tests frequently. This reduces confidence in code quality and impedes agile development approaches that depend on frequent test executions [51].
2. Developers have an incentive to avoid test execution. This has, most likely, a negative effect on software quality. Developers might test less and might introduce changes influenced by testing times.
3. Developers either have to wait two hours or switch tasks. Both scenarios will decrease their productivity.

Test costs affect developer productivity.

In practice, several approaches can mitigate these effects. However, a negative impact on developer productivity will remain. Obviously, these negative effects affect larger projects more significantly than small projects. In summary, the negative effect on developer productivity is another motivation to reduce test costs in terms of time spent on testing.

We conclude that although software quality is important, it is typically not possible to create software without defects. Creating software involves the optimization problem of maximizing software quality while minimizing test costs. While the test costs for small software might be negligible, they are a considerable factor for large projects. Therefore, reducing test costs without affecting quality is an important problem particularly for large projects. In this work, we focus on testing in very large software projects and target the problem of test cost reduction for such projects.

### 1.1 Outline and Contributions

In Chapter 2, we discuss software quality assurance and the problem of test costs with a focus on the specific characteristics of large projects. For this purpose, we study a large industrial project and describe its properties. Additionally, we discuss related work for test cost reduction.

In Chapter 3 and Chapter 4, we focus on coverage, i.e., information about the execution of a program. Chapter 3 provides definitions and technical details of operations and algorithms working on coverage data. The presented algorithms then serve as building blocks for our main contributions in other chapters. In Chapter 4, we discuss the essence and limitations of information gained by coverage data. Furthermore, we provide two empirical studies with new insights into the relationship between coverage and faults. Moreover, we also show a technique of how coverage can be combined with combinatorial testing to achieve a higher testedness with low effort compared to techniques proposed by related work.

In Chapter 5, we analyze several approaches for test cost reductions in large projects. We show that existing work often target small projects and do not consider the specific characteristics of large projects. Furthermore, we present, evaluate and discuss several approaches for test cost reduction tailored to large projects. We also discuss threats to validity of our work.

Finally, in Chapter 6, we focus on our main contribution, dynamic unit test extraction. We present, evaluate and discuss an approach to extract unit tests from system tests via time-travel debugging that combines several techniques presented in this work to achieve practical usefulness. Additionally, we present in this chapter our work for object creation and mock proposal. For both scenarios, we use static analysis to gain insights about object hierarchies and dependencies. We then use this information to propose recommendations to create objects or mocks. In both cases, we present algorithms to minimize the amount of required objects or mocks.

We conclude our work in Chapter 7. We summarize our results, set them in context, and provide directions for possible future work.

The main contribution of this work is an approach for dynamic unit test extraction via time-travel debugging, a practical approach to reduce test costs by extracting unit tests from system tests. The approach is practical because it is implemented for a large industrial software project and it is designed in such a way that it fulfills several requirements of practitioners.

The following list contains our contributions. All contributions are in the context of large projects. Due to brevity and repetitiveness, we omit to name this specifier for each entry.

1. Dynamic unit test extraction via time-travel debugging, a practical approach to reduce test costs by extracting unit tests from system tests via backward-execution of a test execution (Section 6.1).
2. An empirical study of the impact of coverage on bug density (Section 4.2).
3. An economic approach for test case selection and reduction (Section 4.2).
4. An approach for recommending options to create objects in C++ that aims to reduce the total amount of required intermediate objects (Section 6.2).
5. A study of Granger-causality between coverage and defects (Section 4.3).
6. A technique that combines coverage and combinatorial testing to create new tests with low effort (Section 4.4).
7. An analysis of compression techniques for coverage data (Section 5.4).
8. An analysis of causes for test costs (Section 2.1).
9. An analysis of approaches for automated mock recommendation and mock minimization in C++ (Section 6.3).

This thesis includes work that is already published or is planned to be published. For all work in the following lists, the author of this thesis is the main author and is responsible for the conception and design of the studies and techniques, the acquisition of data, the evaluation, the analysis and interpretation of results, and has drafted and finalized the articles. However, the author of this thesis is thankful to all co-authors that also contributed substantially to the aforementioned tasks.

1. Thomas Bach, Artur Andrzejak, and Ralf Pannemans. 2017. Coverage-Based Reduction of Test Execution Time: Lessons from a Very Large Industrial Project. *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2017)* [16].  
DOI [10.1109/ICSTW.2017.6](https://doi.org/10.1109/ICSTW.2017.6)
  - Partially included in Chapters 3 and 5.
2. Thomas Bach, Artur Andrzejak, Ralf Pannemans, and David Lo. 2017. The Impact of Coverage on Bug Density in a Large Industrial Software Project. *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2017)* [17].  
DOI [10.1109/ESEM.2017.44](https://doi.org/10.1109/ESEM.2017.44).
  - See Section 4.2.
3. Thomas Bach, Ralf Pannemans, and Sascha Schwedes. 2018. Effects of an Economic Approach for Test Case Selection and Reduction for a Large Industrial Project. *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2018)* [20].  
DOI [10.1109/ICSTW.2018.00076](https://doi.org/10.1109/ICSTW.2018.00076).
  - See Section 5.2.
4. Thomas Bach, Ralf Pannemans, Johannes Haeussler, and Artur Andrzejak. 2019. Dynamic Unit Test Extraction via Time-Travel Debugging for Test Cost Reduction (short paper). *41st International Conference on Software Engineering (ICSE 2019)* [19].  
DOI [10.1109/10.1109/ICSE-Companion.2019.00093](https://doi.org/10.1109/10.1109/ICSE-Companion.2019.00093).
  - See Section 6.1.
5. Thomas Bach, Ralf Pannemans, and Artur Andrzejak. 2020. Determining Method-Call Sequences for Object Creation in C++. *IEEE International Conference on Software Testing, Verification and Validation (Porto, Portugal) (ICST 2020)* [18].
  - See Section 6.2.

Unpublished:

1. Thomas Bach and Ralf Pannemans and Johannes Haeussler and Artur Andrzejak. Dynamic Unit Test Extraction via Time-Travel Debugging for Test Cost Reduction (full paper)
  - See Section 6.1.
2. Thomas Bach and Ralf Pannemans and Artur Andrzejak. Automated Mock Recommendations.
  - See Section 6.3.

In the following cases, the author of this thesis is not the main author:

1. Artur Andrzejak and Thomas Bach. 2018. Practical Amplification of Condition/Decision Test Coverage by Combinatorial Testing. IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2018) [9].  
DOI [10.1109/ICSTW.2018.00070](https://doi.org/10.1109/ICSTW.2018.00070).
  - See Section 4.4. Contributions: Coverage-based analysis, test suite reduction, implementations, evaluation, partially written the article.

In submission or review:

1. Pavneet Singh Kochhar and Thomas Bach and Gede Artha Azriadi Prana and Ralf Pannemans and Artur Andrzejak and David Lo. A Longitudinal Study on Coverage and Test Suite Effectiveness.
  - See Section 4.3. Contributions: Acquisition of data for SAP HANA, design and discussion of experiments, implementations, discussion of Granger-causality, partially written the article.



# 2 | Background

In this chapter, we discuss the background and context of this work. We introduce software quality assurance and describe the problem of test costs for quality assurance activities. We further introduce research areas related to the problem of test cost reduction and discuss related work. We also present our main research subject, SAP HANA, a large industrial software system developed by SAP. Furthermore, we provide details about the testing process and the resulting test costs of SAP HANA.

## 2.1 Software Quality Assurance and Testing

We start with a simple introduction based on an example and then provide a more detailed discussion of the important aspects.

*Software quality assurance (SQA)* is a set of activities that ensure a certain degree of quality for a software. In a simplified view, quality assurance is a repeated process consisting of the two activities definition and measurement. As a recurring example, we define the following quality standard:

Software quality assurance

*Quality standard:* Each functionality in a software must be documented.

To assure the quality, we measure for each functionality whether it is documented or not. We conclude that the software fulfills the quality standard if all checks are positive. Otherwise, in the case documentation is missing, we conclude that the software does not fulfill the quality standard.

Considering the case that software does not fulfill a quality standard, we typically not only state this fact, but we act to change the state. In our example, such an activity could be the addition of new documentation or the removal of undocumented functionality. After the improvement activity has finished, we measure the current state again and compare it against the quality standard. We can repeat this process until the standard is fulfilled. We can apply this process not only in cases where a quality standard was not fulfilled. It is even possible to further improve quality above a certain standard. For our example, we may verify whether the documentation is complete, correct or helpful to developers. We then iteratively improve the documentation until the higher quality degree is reached.

This set of tasks (plan, do, check, act<sup>1</sup>) provide the foundation for quality management [136]. In contrast to SQA, *software quality management (SQM)* aims to manage, develop and improve the quality of software. While SQM is an important topic, it is also a broad topic [136] that we do not cover in this work. Instead, we focus on SQA and discuss important aspects of SQA.

<sup>1</sup> Also called PDCA circle or Deming circle [63]. The steps are similar to the scientific research process: hypothesis, experiment, evaluation, analysis. Software quality management (SQM)

SQA is typically applied with a varying degree of formalization for all software projects. Personal experience tells us that even a small software program quite often does not work as intended in the very first version and requires further corrections. In such a case, the quality standard is our personal expectation of the required functionality. The measurement is the execution of a program and the improvement activity is the modification of the source code until the execution of the program results in an expected result. Larger software projects and professional software projects may formalize software quality assurance so that the standards and activities are defined and verified. Such a formalization provides confidence that the software quality is appropriate [136].

Software testing is a measurement activity within SQA. A typical implementation of quality measurement for software is the execution of the software and verification of the result corresponding to the execution. More formally, “software testing consists of the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain.” [233]. In contrast to these *dynamic* techniques, “static techniques are different from and complementary to dynamic testing. (...) It is worth noting that terminology is not uniform among different communities and some use the term testing also in reference to static techniques.” [233]. The term *static testing* typically refers to tests that do not require the execution of a software. For example, they check the correct usage of programming idioms. For the rest of this thesis, we do not differentiate between dynamic and static testing (or verification). We assume that testing contains both dynamic and static techniques, but we typically focus on the execution of tests and dynamic techniques.

Testing requires a *test oracle*. That is, a mechanism or source of information to decide whether an outcome is correct or not [21, 125]. For example, we may execute a square root function with an argument of 100. The result is 10. Common sense implies that this result is correct. However, a CPU typically does not provide common sense, it requires instructions. Therefore, a test must encode 10 as the expected result and must verify whether the calculated result is identical to the expected result. Fig. 2.1 shows a typical implementation of such a test.

Depending on the test type, we may encode the oracle within an automated check or we have to verify the results manually. For example, graphical output often requires manual inspection. In some cases, the test oracle may not be stated explicitly but can be derived implicitly. For example, the absence of exceptions or crashes may represent implicit oracles. In these cases, the presence of such an event results in a test failure. In regression testing, an oracle can also only verify that a result is identical to the result of a previous version.

The *oracle problem*, i.e., determining an oracle for a specific input, can be arbitrarily complex. For instance, determining an oracle for the input 101 to a square root function requires a precise understanding of floating point arithmetic [102, 197]. Determining an oracle for the input  $-3$  requires a decision on functionality provided by the square root function, i.e., whether it works on real or complex numbers. Given these examples, we conclude that the oracle problem can be challenging.

ISO 9000 [136]

Software testing consists of dynamic verification.

Test oracle

```

1 @Test
2 public void testSqrt() {
3     int input = 100;
4     int expected = 10;
5     int result = (int) Math.
        sqrt(input);
6     assertThat(result).
        isEqualTo(expected);
7 }

```

Figure 2.1: A typical fluent unit test in Java based on junit [187, 240] and AssertJ [64, 176]. The oracle consists of the expected result and the comparison against the actual result.

Oracle problem

### 2.1.1 Test Organization

Software testing consists of several categorizations that help structuring testing activities. For example, the *test level* refers to the target of the test and the objectives of the test. In addition, tests are grouped at different levels. A single test may be called *test case* and a set of test cases may be called *test suite*. A test suite may be distributed over several files, or a software may have several test suites for different testing objectives. The test level and test objectives provide a characterization of tests. Based on this characterization, different testing purposes might only require a subset of all tests. For instance, performance tests require a different kind of tests compared to functionality tests. Even in within the group of functionality tests, we can select different subsets depending on the specific functionality that should be tested. For example, we can separate a software project into components and only select functionality tests that are relevant for a component that is tested. This concept leads to techniques for test reduction that we discuss in Section 2.1.4.

Based on the *test target*, we can categorize tests into groups such as unit tests, component tests, integration tests, and system tests. A unit test targets a function or unit level, a component test targets a component or class, and an integration test targets the interaction between units, components, functions or classes. Finally, a system test targets the functionality of the complete system. We call them different *layers of testing*. Each layer, in the presented order, typically executes a larger amount of code compared to the previous layer. Therefore, a higher layer relates to a longer execution. On the other hand, tests at the lower layer test only a very specific functionality where at the higher layer they test overall functionality which may be more relevant for a user. They also provide a different level of detail for root cause detection. A fault identified by an unit tests might be faster to find compared to a fault identified to a system tests. In addition, the developer effort to create tests on each level might be different. However, this depends on the specific setting of a software project. Table 2.1 summarizes the characteristics in terms of scope, execution time and amount of tests.

Test Layer	Scope	Execution Time	Amount of Tests
Unit	Functions	Short	Large
Component	Classes	Short to medium	Medium
Integration	Interactions	Short to high	Medium to small
System	Whole system	Long	Small

Test level

Test case

Test suite

Test target

Layers of testing

Table 2.1: Overview of test layers and their characteristics.

The *test objective* allows for categorization of tests based on their purpose. For example, performance and stress tests verify whether a software executes within a predefined time window or how the software behaves if used by multiple users concurrently. Security testing investigates whether functionality or data can unintentionally be accessed. Regression testing verifies whether a software continues to work as expected after a change. Acceptance testing verifies whether the requirements are met. Usability testing measures how real users use a system and investigates the ease of use for a software. The literature provides a wide range of test objectives [133, 233].

Test objective

### 2.1.2 Test Techniques

*Test technique* refers to the systematic approach that is used to conduct the test. They can be grouped in categories such as input-domain based, code or dataflow based, fault based or model based. Related work propose a wide range of different techniques with different goals [43, 208]. We focus on the group of input-domain based techniques. Such a technique selects an appropriate set of inputs for a given test objective. It might be tempting to thoroughly and exhaustively test the full input space. However, even for small programs, this is typically not possible as we will show by an example.

Test technique

Assuming hardware that can execute  $3 \times 10^9$  operations per second representing a 3 GHz processor and assuming one value verification requires one operation. Then, a single boolean input requires two operations with an execution time below a second. For a 32 bit integer input, we must iterate  $2^{32}$  input values. It already requires 2 seconds to execute these operations, but is still feasible. However, a 64 bit integer input requires  $6 \times 10^9$  seconds or at least 190 years. Even with additional hardware, we may not be able to finish testing in a reasonable amount of time. In practice, there can be multiple inputs and the type of the input can be of an arbitrary length such as text from filesystem or data in memory. Therefore, the required execution time can be arbitrarily long. In addition, even in the case we could iterate over all inputs, we still have to verify the output. In conclusion, it is often not feasible to test the complete input space.

Testing the complete input space is typically not feasible.

There are several techniques to reduce the input space for input-domain approaches such as equivalence partitioning, pairwise and combinatorial testing, boundary value testing and random testing [208, 235]. Each term nowadays represents an own field of research with different approaches. We highlight the general principles behind each approach and provide information about related work for further information.

#### 2.1.2.1 Boundary Value Testing

For *boundary value testing*, we select values from the input space where we expect a boundary for a condition. We find such conditions from requirements or source code. For example, a function should return 0 for all values larger than 5. Here, we select 5 and 6 as boundary values for testing. Empirical research shows that such an approach is effective [49, 88, 90, 219, 235].

Boundary value testing

#### 2.1.2.2 Equivalence partitioning

*Equivalence partitioning* (EP) is a standard approach for test input generation [88, 109, 235]. Based on the specification, we categorize the input domain into one or more sets where each item in a set shows the same behavior for the program execution. We call each set an equivalence class (EC) [230]. Then, for each EC, we test only a single representative<sup>2</sup>. We can combine EP with boundary value testing, i.e., we select three candidates for each EC: the “minimal” item, a random item, and the “maximal” item. For example, to test the function  $abs(x) = \sqrt{x^2}$  for the input domain  $[-10, 10]$ , we define the two EC  $\{[-10, -1], [0, 10]\}$ . For testing, we select the representative input values  $\{-10, -7 - 1, 0, 2, 10\}$ . EP reduces a large

Equivalence partitioning

<sup>2</sup> We define and explore the mathematical background of equivalence relations and classes in Section 3.3.8.

input domain to a few elements of representatives. However, there are also limitations. It might not be possible to create EC because of inadequate specification or the problem does not allow the creation of such classes (e.g., creating EC for matrix multiplication). In addition, the amount of EC can be too large for practical usage (e.g., algorithms working with Strings).

### 2.1.2.3 Pairwise And Combinatorial Testing

*Pairwise testing* investigates only combinations of pairs of different parameters in a multi-dimensional input space instead of all combinations of all parameters [164, 165, 195, 203]. This approach is reasoned by empirical studies where root cause analyses of failures show that more than 90% of all faults depend on 2 or fewer variables [166]. Therefore, instead of testing for all  $2^4$  combinations of 4 boolean parameters, it might be sufficient to test only a subset of inputs so that all pairwise combinations between each parameter are represented in the test input. In fact, this requires only 5 tests instead of 16 tests as shown by Table 2.2. The problem of finding such an input is NP complete. However, we might accept also near-optimal solutions. In practice, the difference for test reduction between a large  $2^n$  amount of tests and a small  $n$  is significant, but the difference between the minimum amount of  $n$  and  $n + 10$  tests has probably no measurable effect.

Pairwise testing can provide an exponential reduction for the amount of test cases. For instance, 10 boolean parameters would require  $2^{10}$  test cases to iterate over all combinations, but less than 30 test cases to execute all pairwise combinations [164, 165]. Pairwise testing provides promising results for parameters with small domain sizes such as booleans. For larger domains, such as 32 bit integers or strings, pairwise testing still leads to an infeasible amount of test cases. We can see this by a small discussion of an absolute minimum of test cases that must be generated. For  $n$  parameters where each parameter can have  $m$  different values, pairwise testing must create at least  $m \cdot m$  tests cases, because all pairs of 2 parameters must be executed. For instance, a function with  $n$  32 bit parameters requires at least  $2^{32} \times 2^{32} = 2^{64}$  test cases. As we have already discussed before, this would require a time effort of over 100 years on typical hardware. To tackle this challenge, we can combine pairwise testing with equivalence partitioning as we explore in Section 4.4.

The generalization of pairwise testing is combinatorial testing, where instead of only pairs of parameters, larger subsets and their combinations are investigated [165]. For instance, we may want to cover all 3-way combinations, i.e., all value combinations of all possible subsets with cardinality 3. Empirical studies suggest that extending this approach to 6-way testing may be sufficient for practical scenarios because defects found in practice rarely depend on more than 6 parameters [164, 165].

### 2.1.2.4 Random Testing

*Random testing* uses randomly selected values of the input domain to test the software. Typically, the testing is limited by a threshold in terms of time or number of generated values. The main issue of random testing is the missing test oracle. Generally speaking, random input results in random output.

$P_1$	$P_2$	$P_3$	$P_4$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0

Table 2.2: Pairwise testing example. These 5 tests cover all possible values for all combinations of two boolean parameters.

Pairwise testing

Limitations of pairwise testing

Random testing

Except for implicit test oracles such as crashes and assertions, we have no further knowledge of whether the random output is correct or not. Therefore, random testing has limited value in practice. However, randomly generated tests can be beneficial for differential testing, i.e., comparing whether the output of software (unexpectedly) changes between versions [190]. Instead of generating completely random input, there exist also so-called directed random approaches. There, randomness is used to generate input but the generation is bounded by some constraints concluded from the source of the software or its execution [209].

### 2.1.3 Test Costs

Testing activities require time and resources. It requires time to design, implement, execute and check tests. The development and setup of a test framework may require additional time. As discussed in the previous section, there is a wide range of test techniques for software testing. Given infinite time and resources, we could employ all techniques to ensure a high degree of quality. Unfortunately, time and resources are limited in practical scenarios. This results in a typical cost problem, i.e., how much time and resources should be invested for an expected benefit.

Test cost problem: Limited time and resources.

Also, testing requires resources. Test executions may use a workstation that is also used for development, if the resource consumption of tests is reasonably low. However, test executions with higher resource consumption might require a central system to organize the test executions. A large set of tests might require multiple test servers to allow the execution of all tests within an acceptable waiting time. Moreover, the execution time for tests translates to waiting time for developers. They either wait until testing is finished, or they switch their current task which might add context switching costs. In summary, testing can result in a wide range of costs. Therefore, improving the efficiency of testing is an important task for small and large software projects to reduce test costs.

Test executions require additional hardware and software.

As stated in the introduction, reducing test costs is a trade-off between the effort of testing activities and therefore the degree of quality of a software project on the one hand and associated costs on the other hand. For any non-trivial project, it is typically not possible to precisely model this trade-off due to inaccuracies in calculating the test costs and due to a missing precise definition of quality. There are several approaches to estimate test costs and the impact on software quality [116, 144]. However, it is expected that such approaches must be tailored to the specifics of a given software project and may not be generalizable.

Trade-off between costs and quality.

Even without a precise model, we can still use a simplified model to analyze the trade-off property between test costs and the degree of quality for a software project. Let  $c_t$  be the costs of testing activities and  $c_o$  all other costs. Then the total cost of a software project  $c_{sum}$  follows the equation  $c_{sum} = c_o + c_t$ . Based on the previous discussion, we assume that the degree of quality  $q$  depends on the costs for testing. Therefore,  $q = f(c_t)$  where  $f$  is a monotonically increasing function. We further assume that  $c_o$  is fixed. There exist two options to increase  $q$ . First, by increasing  $c_t$ , i.e., by increasing the test costs, we increase  $q$ . However, this also increases

Simplified model for test costs.

$c_{sum}$ , i.e., the total costs. Second, we can make testing more efficient, i.e., we replace  $f$  by  $f_{new}$  so that  $f(x) \leq f_{new}(x) \forall x$ . This allows us to decrease  $c_t$  while maintaining the same result for  $q$  or increase  $q$  while maintaining the same value for  $c_t$ . In summary, to improve the degree of quality for a software project, we either increase the total costs or we improve the efficiency of the testing<sup>3</sup>.

In practice, it may economically not be advisable to continuously increase the total costs of a software product by proportionally increasing the costs for testing. As we show in Section 5.2, the test costs of a large and successful software may increase superlinearly over time, which makes it infeasible to accept a proportional increase in total costs — even for large companies. In such cases, it may not be possible to guarantee a certain degree of quality due to the high test costs. Therefore, test cost reduction does not only provide a reduction in costs, but may even be required to ensure a certain degree of quality for a large software. Otherwise, the high test costs prevent any further quality activities.

In summary, test cost reduction is important for small and large projects. Even more, test cost reduction can be a requirement to ensure a high degree of quality for large projects, because cost limitations might prevent extensive testing activities if the crated costs are too large.

#### 2.1.4 Related Work on Test Cost Reduction

Test cost reduction is an important problem and has been addressed by researchers and practitioners alike: “Testing can be expensive, and the need for cost-effective techniques has helped it emerge as one of the most extensively researched areas in testing over the past two decades” [208].

Orso and Rothermel [208] use the following three categories:

- *Test case prioritization* (TCP).
- *Regression test selection* (RTS) or *test case selection* (TCS).
- *Test suite reduction* (TSR).

TCP, TCS, and TCR

Yoo and Harman [257] and Hyunsook [76] provide surveys for all categories, Khan et al. on TSR [153], and Khatibsyarbini et al. on TCP [154]. Kazmi et al. provide a literature review about TCS techniques [149] and Ali et al. about the industrial relevance of regression testing research [25].

Each of these three categories describes techniques that filter or reorder test cases to optimize some criteria. For example, we may consider *maximizing code coverage* as the optimization objective, understanding it as a proxy for the thoroughness of testing [103]. TCS techniques select a subset of tests for each test run. For example, only test cases relevant to a specific objective function are selected for a test run. TSR techniques shrink the test suite (i.e., test cases are removed). Therefore, the potential cost savings for TCS and TSR techniques are typically higher compared to TCS, but TCS and TSR techniques have the potential disadvantage of a loss of testing quality because not all tests are executed compared to the full test suite.

The main difference between TCS and TSR is the state of the original test suite. For TCS, the original test suite remains unmodified. Typically, TCS techniques select a subset of tests for a specific activity. TCR techniques

<sup>3</sup> We may also improve other factors that were in fact hidden within the  $t_o$  constant, but these other factors are not in focus of our simplified model. Test costs can increase superlinearly.

modify the original test suite. In this case, instead of the original test suite, only the test suite as a result of TCR will be used in the future.

Techniques of all three categories can be combined in any order. For example, we may first reduce a test suite, then select multiple different subsets of the reduced test suite for different tasks and finally prioritize the tests for each of these tasks. Even more, TCS and TCR techniques are often interchangeable and TCP techniques can be converted to TCS techniques by using a threshold to limit the ordered sequence of test executions.

We provide an extensive discussion of related work for TCP and TCS in Chapter 5 and therefore do not replicate this overview here. Instead, we focus only on more recent work that target large systems.

Menon et al. describe their work at Google [191] where 150 million tests are executed per day. Google engineers implemented a system for TCS. Based on dependency analysis, they collect for each change  $C_1$  the set of required tests  $T_1$  to test this change. Their build system facilitates such analysis, requiring minimal overhead compared to advanced static analysis techniques. Instead of executing all tests in  $T_c$  directly, they collect over a time frame of several hours the set  $T = T_1 \cup \dots \cup T_n$  of all  $n$  tests sets collected within the time frame. Then, they execute all tests in  $T$ . In the case all tests succeed, they conclude that the source code is successfully tested now. In the case one or multiple tests failed, they triage the error-introducing change by re-executing the failed tests for all intermediate changes to find the error-introducing change. This approach consequently reduces test executions by dependency analysis and by merging multiple test executions into a single execution. The approach is effective because the majority of test executions succeed and there is no benefit in executing succeeding tests multiple times. Hence, their approach skips these executions.

Work at Google

Machalica et al. describe their work at Facebook [184] where 10 million tests are executed per day. Facebook engineers implement a system for TCS and TCP. For this purpose, they use machine learning techniques to classify, for a given change, the set of all available tests into potential failing tests based on metadata of the change (such as the size of a change, history, file type, or historical failure rates). Consequently, they only execute tests that are classified as potentially failing. They report that this approach reduced infrastructure costs for testing code changes by a factor of 2, while guaranteeing that over 95% of individual test failures and over 99.90% of faulty changes are still reported back to developers. Most interestingly, the learning based on metadata is language-agnostic, i.e., no further understanding or parsing of a programming language is required.

Work at Facebook

Cruciani et al. describe an TSR approach that aims to be scalable for large systems [57]. They employ similarity-based clustering techniques on test suites and then select only a single test for each detected cluster. They evaluate the execution time of their technique on 500 000 tests collected from several GitHub projects. To evaluate the fault detection loss of their approach, they seeded faults into the source code of study subjects. They conclude that fast execution times and low fault detection loss make their approach practical for large projects.

We conclude that our work is not the first to investigate testing in large software projects. However, as we will see, there are still unsolved challenges.



## 2.2 Study Subject: A Very Large Software Project

Our main case study for this work is SAP HANA [93, 94, 188], a high-performance, parallel in-memory database management system developed by SAP. With over 6 million lines of source code, we consider SAP HANA a very large software project. SAP HANA is mainly written in C++ and C and modified by over 800 source code changes (*commits*) per day<sup>4</sup> by more than 100 developers. The code of SAP HANA is in part very complex due to requirements for high performance, implying own memory management subsystems, and massive multi-threading. The code basis combines and integrates several sub-projects with a lifetime of more than 10 years. The development of SAP HANA creates more than 10 TB of metadata per year that is stored in and analyzed by an internal SAP HANA database itself.

### 2.2.1 Quality Assurance

Since many customers use SAP HANA in mission-critical scenarios, quality assurance of this product is of paramount importance. This requirement is ensured via extensive software testing practices in all development and release stages. To illustrate, there exist over 900 000 test cases that can be executed by over 1 000 servers with, on average, 40 CPU cores, 3 GHz frequency, and 256 GB of memory. Executed sequentially, the total execution time of all tests (even with optimized builds) would require up to 3 weeks.

In practice, tests are executed in parallel on a cluster of test servers. The actual execution times of tests range between 1 min and several hours. The execution time depends on factors such as test type, cluster size, cluster load factor, or test configuration. Effects such as flakiness resulting in re-executions of tests can increase the total execution time.

### 2.2.2 Test Organization

There are three relevant hierarchies for the test environment of SAP HANA: the test layer hierarchy, the test deployment (or execution) hierarchy, and the testing stages hierarchy. The following sections discuss each hierarchy in more detail. In practice, there exist additional layers, but we focus only on the aspects important for our work.

#### 2.2.2.1 Test Layer Hierarchy

The test code of SAP HANA is organized in *test suites*, each containing between 1 to 20 000 *test cases*.

A test suite typically consists of a Python file embedded in a custom testing framework. It can contain different types of tests, mainly: system and unit tests, but also component and integration tests. A system test typically checks for regressions introduced by new changes to the source code. Such a system level regression test is typically implemented as one or multiple SQL queries. The test framework sends each query to a prepared database instance and checks the result for correctness. Contrary to system tests, unit tests call code fragments directly using a C++ test framework and check the results. Counterintuitively to a common understanding that unit

<sup>4</sup>Note that in practice, the metric “changes per day” is rather vague due to the unclear definition of what exactly constitutes a single change. With pre-commit reviews, even the number of commits or the number of merges to the main repository may not be representative. In our case, we count the number of changes to the main product lines and each branch that represents a component within SAP HANA. However, such a single change can be the result of multiple and iterative smaller changes.

900 000 test cases

3 weeks sequential test execution time

Focus on system and unit tests.

tests execute a small amount of code, there are also unit tests that require a database instance and therefore execute the whole database setup and database stack and are still referred to as unit tests by (some) SAP developers. It seems that historically, some developers considered all C++ tests as unit tests and all Python tests as system tests.

As we can already conclude, the exact categorization of tests into test layers is not trivial in practice. There exists a definition for each layer [134, 233], but they can contain ambiguity or are not practical [86]. Based on discussions with developers at SAP, we recognize that developers have different interpretations of these terms, which aligns with related literature regarding the differentiation between unit tests and integration tests [86]. In this work, we typically only differentiate between system tests and unit tests. We categorize tests that use SQL statements and the Python test framework as system tests and tests that use C++ and any of the existing C++ test frameworks as unit tests. We see in Section 5.5, that we can also use coverage data to effectively differentiate such tests.

Python: System test. C++ : unit test.

### 2.2.2.2 Test Deployment Hierarchy

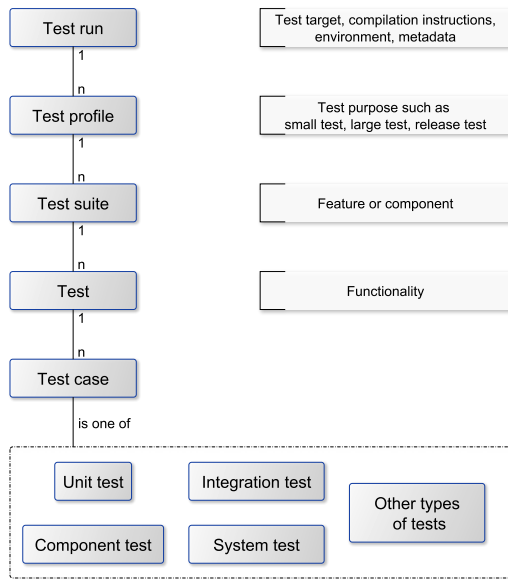


Figure 2.2: The organization of the test deployment hierarchy SAP HANA. Each level consists of multiple refinements.

The test deployment hierarchy determines which test cases are grouped and executed together. Fig. 2.2 shows an overview with descriptions.

Starting with a *test run* that identifies a quality review for a specific version of the software, each level in the deployment hierarchy contains one or multiple refinements. A test run consists of a set of *test profiles* that group test suites by their test purposes. Such a test profile can correspond to a project component (source module), or it can organize test suites by other criteria. A test profile contains *test suites* that target a set of features or components. Each test suite contains a *test* layer that only simplifies the organization of *test cases* and different types of test cases. We therefore typically omit the test layer in further discussion. A test case then represents a specific single test and can be of different types.

Test run  
 Test profiles  
 Test suites  
 Test cases

Developers can also execute single test cases or test suites either on a local development workstation or on the central testing system. The central testing system utilizes either test servers directly or they use virtualization techniques such as docker environments [29, 192].

SAP uses the SAP HANA database itself to store the results and metadata of all test runs for convenient access and self-testing for SAP HANA.

### 2.2.2.3 Testing Stages Hierarchy

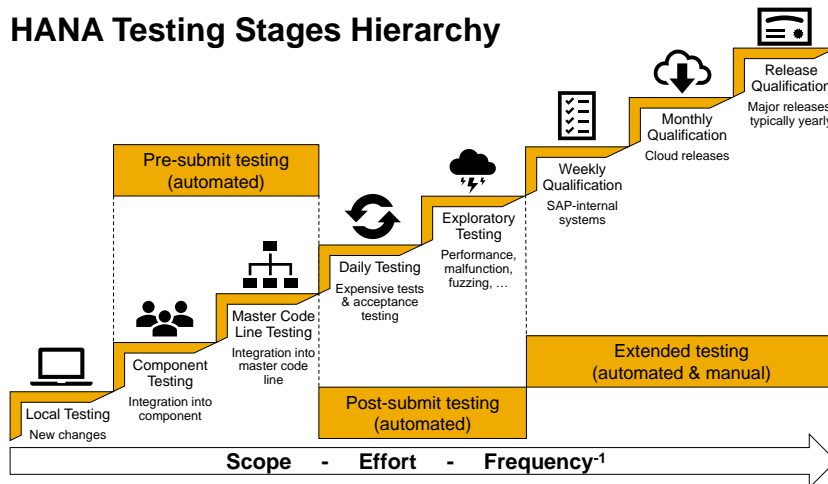


Figure 2.3: The testing stages of SAP HANA. Source: Sascha Schwedes / SAP (modified for presentation).

The quality assurance process for SAP HANA is divided into several testing stages as shown by Fig. 2.3. Broadly speaking, lower testing stages have a smaller scope, run shorter, require less hardware, and run more frequently compared to higher testing stages. The stage of local testing is rather informal. It is the responsibility of a developer to test locally (or not) and there is no metadata available for this stage, i.e., it is unknown which tests are run how often and whether they succeed or not. Developers can also execute tests of a higher testing stage, but they may also apply approaches that are not contained in higher stages. Starting from component testing until the highest testing stage, all stages involve automated parts and SAP records metadata for all stages. In several parts of this work, we utilize this metadata to gain insights about the testing process.

The existence of multiple testing stages has several consequences of which we only list a subset that is important for our work:

1. The existence of multiple testing reduces the feedback time for developers. Developers do not have to wait until all tests are executed, they gain early feedback based on a smaller subset of tests.
2. Developers might not get immediate feedback on their changes if a fault is only found in one of the later testing stages.
3. The resource consumption is reduced. As higher testing stages typically involve more expensive tests, the reduced execution frequency results in reduced costs compared to executing them for every change.
4. It requires a substantial amount of effort to keep changes, tests, versions and bug reports in synch.

SAP records metadata for testing.

All these consequences either improve the productivity of developers or need additional measures to avoid negative effects for the degree of quality. We further investigate the concept of multiple testing stages in Section 5.2.

### 2.2.3 Test Costs

As already mentioned in Section 2.2.2, executing the set of 900 000 test cases can require a considerable amount of time and resources such as hardware. In addition to the issue of a large number of tests, there are several other aspects that increase test costs such as tests with long execution times, high execution frequency, waiting times for test executions, or follow-up activities to verify the test result. We briefly discuss these aspects.

#### 2.2.3.1 Large Amount of Test Cases

We consider the amount of 900 000 test cases a fairly large number that is hard to inspect individually. However, we can categorize test cases by the different criteria discussed in Section 2.1. This allows us to identify subsets of particular interests. For instance, unit tests have a lower execution time compared to system tests. The former typically execute in a fraction of a second whereas the latter can require more than an hour to execute. Therefore, in terms of execution time, a single unit test has a considerably lower impact on the test cost compared to a single system test. Generally, we expect that tests in a higher testing layer require more execution time compared to tests in a lower testing layer. In addition, the test purpose can also provide a characterization with similar implications. Performance tests or fuzzing tests with some type of random execution typically require more execution time compared to a functional test. For this discussion, we characterize tests that require a long execution time for the execution or extensive resources (such as distributed tests) as expensive.

Not all tests are equally expensive.

We also observed that the amount of test cases grows over time and the deletion of tests rarely occurs. Based on discussions with developers at SAP, we believe that the reason for this observation is based on insufficient incentives for developers to delete tests. Even though a test run could be shorter, tests are executed centrally and a developer might not even notice a small change in the execution time. On the other hand, test deletion requires time and might result in a test suite with a reduced degree of quality. Moreover, developers do not want to be responsible for deleting a test that would have detected a fault in future versions of the software. Conclusively, developers write and add more tests over time and rarely delete tests. Therefore, the amount of test cases grows over time.

The amount of tests only grows over time.

Given a large number of test cases, we have multiple options to improve test costs. We can either remove tests, we execute tests less frequently, or we focus on non-expensive tests if new tests are added. We noticed that the first option of test deletions is not practical and the last option is rather speculative based on future improvements. Therefore, our work explores the second option in Chapter 6, where we propose to extract unit tests from system tests and displace the system tests to a higher testing stage with lower execution frequency and thereby reducing test costs.

### 2.2.3.2 Tests with Long Execution Times

For SAP HANA, a large fraction of automatically executed tests can be categorized into one of the two categories unit tests or SQL system tests. Regarding execution time, i.e., the time between the start of a test execution by the central continuous integration system until the results are reported, these two categories have different characteristics as shown by Table 2.3. For this reason, we label unit tests as *short* or *fast* and SQL system tests as *long* or *slow*. In practice, there are also unit tests with higher execution times than some system tests, but for the majority of cases, these labels fit.

Unit tests: fast. System tests: slow.

Required Time		Task
Unit Test	SQL System Test	
$\geq 2$ s	2 h	Compile C++ source code to binaries
	5 s to 2 h	Schedule for free server resources
$\geq 5$ min	$\geq 30$ min	Distribute binaries
-	$\geq 20$ min	Install HANA
-	$\geq 1$ min	Start HANA
-	$\geq 10$ s	Send SQL statement
$> 0$ s	5 s to 3 h	Execute and wait for result
	5 min	Check result, report to CI
$< 30$ min	Several hours	Total required time (typically)

Table 2.3: Composition of test execution times for tests executed by the central continuous integration system. Outliers are possible and occur frequently, however, we omit them to simplify the presentation.

Given these aspects, a single SQL system test contributes a considerably larger part to the total test costs compared to a single unit test. Therefore, we focus our work in Chapter 6 on such SQL system tests.

### 2.2.3.3 High Execution Frequency of Tests

At SAP HANA, developers commit on average more than 800 changes per day to the central source code repository. Each of these commits requires *pre-commit testing*. Therefore, the test costs depend on the number of commits. Furthermore, the number of commits depends on the number of developers. As large software projects such as SAP HANA typically employ a comparatively large amount of developers, the number of commits is also considerably large as shown by the number of 800 commits per day. In practice, each of these commits can already consist of several smaller changes. In conclusion, a large amount of developers results in a high frequency of commits and therefore in a high frequency of pre-commit testing that leads to large test costs. Section 5.2 provides further discussion of this issue.

A large amount of developers leads to large test costs.

### 2.2.3.4 Waiting Times for Test Execution

As stated before, developers commit more than 800 changes per day for SAP HANA. Each change is tested before the change is merged into the central code repository. As shown in Section 2.2.3.2, a test run can require several hours of execution time. Assuming that a test run requires only one hour of time, then 800 changes and therefore 800 test runs would require 800 hours if executed sequentially. The sequential execution is required to guarantee the correctness of the system after each change. However, a

A day has not enough hours to execute all required tests.

single day has less than 800 hours. As a result, it is not possible to execute all test runs sequentially. Either test runs must be executed in parallel or the development process must be adapted to reduce the number of changes so that testing can be finished before the next change is committed. For the development of SAP HANA, testing is done in parallel. Each time a change is committed to the central source code repository, a new test run is executed to test the change. The change is merged if testing succeeds.

However, parallel testing can not guarantee that the final state of the source code passes all tests after each tested change is merged. Suppose that change *A* implements new functionality that uses an object *O* and change *B* modifies the behavior of *O*. Both changes pass separately all tests. However, the combined source code can contain defects due to different logic implemented in both changes. Such conflicts are not detected as merge conflicts by the version control system. We call this group of conflicts *logical merge conflicts*. Related work provides several approaches to detect and solve logical merge conflicts [7, 35, 112].

### 2.2.3.5 Test Flakiness

Test flakiness describes the effect that a test passes and fails in multiple executions for the same code. We discuss in Section 5.6.2 the problems that flaky tests create. A typical approach for tackling test flakiness is to rerun failed tests. Each time a test fails, we execute it  $n$  additional times. If the test succeeds at least once in these  $n$  executions, we conclude that the test is flaky. We may either ignore it or further investigate it. These additional test executions increase the test costs. Given a factor  $p$  of flaky test and a set of  $m$  tests, we statistically have to execute  $p \times m \times n$  additional test executions. Based on related work, typical values are  $p = 0.1$  and  $n = 5$ . Therefore, we expect a test cost increase by a factor of 0.5.

Additional test executions caused by flakiness.

In addition to the test cost increase by additional test executions, test flakiness also affects other aspects. The approach described above represents a happened-before relationship, i.e., the additional test executions can only start after a test failure occurred. Therefore, the total time for executing all tests can substantially increase if a test with a rather large execution time fails at the end of a test run.

Total test execution time increases.

Flaky test results also require additional developer effort. It is typically unclear whether the source of test flakiness is related to the tested software or to the testing environment. It may happen that the software contains defects that appear only with a certain probability for each execution. Therefore, additional analysis by developers might be required. Even more, the analysis of such defects might be rather time-consuming as a developer might not be able to easily reproduce the issue.

Developer effort increases.

The effects of flakiness are more visible in larger projects. Let a small project have 100 tests. A flakiness rate of 0.001 for all tests would result in one random failed test every 10 test runs. Developers might even ignore such a result if they rerun the test suite and all tests succeed. In contrast, a large project with 100 000 tests shows several failed tests for every test run.

A full test run rarely succeeds due to flakiness.

In conclusion, test flakiness can, among several negative effects, considerably increase test costs and is more visible in large projects.

### 2.2.3.6 Conclusions to Test Costs

Test costs typically exist for all software projects. However, the properties of large projects can make effects visible that go unnoted in small projects. The large number of tests, the long execution times and the high frequency of test executions considerably increases the test costs for large projects. Additional factors, such as test flakiness, waiting times for developers and time required for follow-up activities further increase the test costs.

### 2.2.4 Code Coverage Data

Developers at SAP HANA use *DynamoRIO drcov* [67] to regularly collect line-based code coverage data for test executions, i.e., information whether each particular source code line was executed (“hit”) or not during a test run. It would be surely interesting to use a more fine-grained granularity than line coverage (i.e., on branch, statement or instruction level), but previous internal studies showed that collecting coverage data with more fine-grained granularity than line coverage considerably increases test execution times and size of coverage data. Therefore, line coverage is used as a suitable compromise between accuracy and resource usage. Even for line coverage, the cumulative execution time of tests with enabled instrumentation for collecting coverage data is about 1 877 hours or 78.20 days – an increase by a factor of 6. A typical coverage run still requires up to 2 days if executed in parallel on multiple servers. Moreover, each such coverage run generates about 130 GB of code coverage data. We further define and discuss the terms coverage and related problems in Chapter 3.

Line coverage for SAP HANA.

## 2.3 Summary

Software quality assurance is an important task for software development, but it also can require considerable efforts in terms of time and cost for testing. We identify several factors contributing to high test costs that are unique or only significant for large projects. However, high test costs also provide an opportunity for large test cost reduction which shows the relevance of research in test cost reduction.

There is a wide range of research for test cost reduction that aims to reduce the effort required for testing while minimizing reductions in the degree of quality. As our discussion shows, several approaches proposed by researches and practitioners have limitations for large software projects or even do not consider the specific properties of large projects. Therefore, our contributions aim to reduce this gap. We investigate several aspects in the following chapters. Section 5.6.2 investigates approaches for test cost reduction in large projects and Chapter 6 introduces our main contribution to reduce the costs of tests with long execution times.





# 3 | Code Coverage: Measure Test Execution

Software tests execute a program, the software under test (SUT). Such a program execution consists of a sequence of instructions interpreted by a processor. We can monitor the instructions that are actually executed by the processor to obtain *coverage data*, i.e., information about the execution of the SUT. Furthermore, by mapping the instructions to source code, we can measure the *code coverage*. The code coverage allows us, for a given SUT, to identify the subset of source code that is executed by a test (“covered”) and the complementary part that is not executed by a test (“uncovered”).

In this chapter, we define coverage and briefly explain the collection process. We discuss multiple problems related to the analysis of coverage data and present algorithms to solve them. These algorithms serve as a foundation for advanced techniques in other chapters. We discuss the implications and limitations of information gained by coverage data in Chapter 4.

## 3.1 Definitions

Code coverage is frequently used in practice and research [6, 46, 83, 120, 122, 140, 202, 213, 248, 256]<sup>1</sup>. However, there exist different variants of coverage and the exact meaning of the term coverage may depend on the context. For instance, a comparative study of 9 code coverage tools shows that the results for multiple coverage criteria differ significantly across the different tools, even with cases where some tools show 100% and other tools 0% coverage for the same case [202]. Therefore, it is important to define the exact meaning of coverage and related terms. In this section, we discuss the context and introduce definitions for multiple variants of coverage.

### 3.1.1 Context

A *computer program*  $P$  consists of a sequence  $S_{\text{instr}}$  of instructions that can be interpreted and executed by a computer as shown by Fig. 3.1. We focus in our work on the x86/x64 architectures and, consequently,  $S_{\text{instr}}$  consists of x86/x64 instructions [1, 2, 131, 132]. Instead of the term computer program, we may also use the terms software or software program.

A developer that creates  $P$  can provide  $S_{\text{instr}}$  directly. However, instead of writing x86/x64 instructions, a developer typically uses a formal language to write a sequence of words and characters that will be translated into  $S_{\text{instr}}$  as shown in Fig. 3.2. Such a formal language is called a *programming language* and the sequence of instructions  $S_{\text{code}}$  written in such a program-

Coverage records what was executed.

<sup>1</sup> A literature search for “code coverage” finds more than 100 publications. A survey from 2009 provides an overview [256].

```
1 0000000400400 <main>:  
2   imul $0xabcd,%edi,%edi  
3   jmpq 400500 <_Z6sqi>  
4 0000000400500 <_Z6sqi>:  
5   mov %edi,%eax  
6   imul %edi,%eax  
7   retq
```

Figure 3.1: x86/x64 instructions.

```
1 __attribute__((noinline))  
2 int square(int number) {  
3     return number * number;  
4 }  
5 int main(int c, char*[]) {  
6     return square(c*0xabcd);  
7 }
```

Figure 3.2: Source code.

Programming language

ming language is called *source code* or just code [114, 245].  $S_{\text{code}}$  consists of  $n$  files organized within a file system and each of these  $n$  files consists of 0 to  $m_i$  lines,  $n \in \mathbb{N}^+$ ,  $m_i \in \mathbb{N}_0 \forall i \leq n$ . The content of files consists of characters typically encoded in ASCII [10] or UTF-8 [137, 246]. The content is organized in lines. The placement of line breaks, and therefore the amount of lines, is determined by the developer. In practice, developers do not introduce line breaks at arbitrary places, but aim to improve the readability of the source code for humans. Therefore, developers follow a common understanding or a style guide [152] for placing line breaks.

Source code

The translation from  $S_{\text{code}}$  to  $S_{\text{instr}}$  provides information about the mapping  $M \subseteq S_{\text{code}} \times S_{\text{instr}}$  between a specific part of  $S_{\text{code}}$  and the corresponding subset of  $S_{\text{instr}}$ . That is, we know for each element  $i$  in  $S_{\text{instr}}$  which subset of  $S_{\text{code}}$  creates  $i$  and we know for each meaningful part of  $S_{\text{code}}$  which instructions will be generated for this part. It is possible that some lines of the source code map to an empty sequence of instructions, for instance, in the case of source code that is ignored during translation or for empty lines. Vice versa, instructions in  $S_{\text{instr}}$  can map to an empty set of characters in the source code, for instance, in the case of implicitly generated instructions.

Mapping between source code and instructions.

### 3.1.2 Instruction Coverage

For the execution of a program  $P$ , a computer executes the instructions in  $S_{\text{instr}}$ . However, not all instructions of  $S_{\text{instr}}$  may be executed because x86/x64 allows arbitrary jumps between them. The term *instruction coverage* refers to the sequence of instructions  $S_{\text{IC}}$  that are executed for a specific execution of  $P$ . More specifically, the instruction coverage  $IC_{PE}$  for a program execution of  $P$  is a sequence of numbers  $\langle h_1, h_2, \dots, h_{|S_{\text{instr}}|} \rangle$  where  $h_i$  represents the amount of times the instruction at position  $i$  was executed (“hits”). For practical efficiency reasons,  $h_i$  may be restricted to  $h_i \in \{0, 1\}$ , indicating only whether an instruction was executed or not. In addition, the cases where  $h_i = 0$  may not be contained in  $IC_{PE}$  and all items in  $IC_{PE}$  have an additional label that indicate the corresponding instruction. Furthermore, we define the *instruction coverage ratio ICR* as

Instruction coverage

Boolean counter.

Instruction coverage ratio

$$ICR = \frac{|\{x \mid x \in S_{\text{IC}} \text{ and } x \neq 0\}|}{|S_{\text{instr}}|}. \quad (3.1)$$

A specific execution of  $P$  may depend on additional input to the program such as the state of memory. We assume that such input does not change for multiple executions of the same program if not stated otherwise. Therefore, we expect that a program execution is deterministic, i.e., two executions of  $P$  will always produce the same instruction coverage for the same input.

Assumption: Deterministic program execution.

The assumed deterministic model of computation simplifies the theoretical discussion. However, it does not reflect a practical environment. Several aspects, such as concurrent execution, random number generators, time, or input from the filesystem or users may result in a scenario where two, seemingly identical, executions produce different instruction coverage. We further discuss these practical issues in Section 5.7 and show there that we can “normalize” such coverage data, i.e., we can make it appear deterministic. Therefore, our simplification is reasonable.

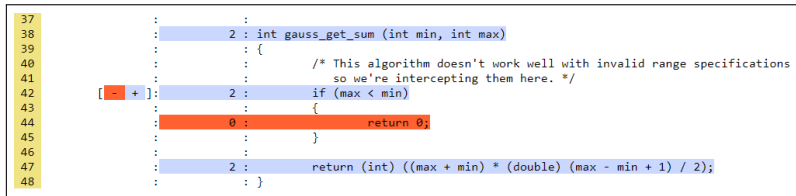
### 3.1.3 Code Coverage

We obtain the instruction coverage  $IC_{PE}$  for a program execution  $PE$  of  $P$  as described in Section 3.1.2. We map the  $IC_{PE}$  to  $S_{code}$  via the mapping  $M$  provided by the translation of  $S_{code}$  to  $S_{instr}$  as described in Section 3.1.1. The result is the *source code coverage*  $CC_{PE}$ , or *code coverage*, a sequence of numbers that indicate for each part of the source code how often this part was executed by a specific execution of  $P$ . More formally,

$$CC_{PE} = \langle s_1, s_2, \dots, s_{|S_{code}|} \rangle \tag{3.2}$$

$$\text{where } s_i := \sum_{j \in \{y | \forall x_y \in S_{instr} : (y, i) \in M\}} h_j \tag{3.3}$$

The term code coverage may be used in two different contexts. It may represent a) coverage data, i.e., information about the execution of the source code, or b) a percentage, i.e., the amount of executed source code relative to the total amount of source code. Fig. 3.3 shows a visual representation of coverage data where executed areas are highlighted.



Source code coverage

Figure 3.3: Visual representation of coverage data. Numbers indicate how often a line was executed. Not executed lines with orange background. Lines that do not map to instructions with white background. The actual source code is not relevant. Source: LCOV documentation [15].

The code coverage is always obtained for a specific execution of the program. However, the specific execution context may not be provided. In such cases, code coverage typically refers to the execution of tests and we identify the corresponding code coverage with the term *test coverage*. The coverage data may contain only coverage information for the code under test, but may also include coverage information for the test code.

Coverage relates to a specific execution.

We assume a fixed set of tests for the generation of test coverage. In practice, it may be unclear which tests are considered for test coverage and the set of tests changes over time. Test coverage may be obtained by the execution of all tests in a single execution or by the execution of every single test separately, possibly followed by an aggregation step.

Test coverage

### 3.1.4 Data Coverage

Instruction coverage contains information whether, e.g., an `if` condition statement was executed or not. However, we have no further information about the result of the execution (*true* or *false* for the `if`) or the cause of the results if the statement depends on a variable (such as `if (x<10)`). *Data coverage* contains information about the input or output of operations that consists of instructions in  $S_{instr}$ . An instruction operates on a fixed instruction data width with up to 512 bit for *AVX-512* [132]. Therefore, the input and the output of an instruction can be any of  $2^{512} \approx 10^{154}$  bit combinations. In addition, an algorithm can depend on a possibly unbounded amount of instructions. Such unbounded cases can appear for loops and string processing. In conclusion, the domain of the data coverage

Data coverage

is in the general case unknown and can be rather large or even infinite. In practice, however, types such as a 32 bit integer or a 2 bit boolean are small enough to enumerate the whole domain. Given such considerably small finite domains, we can calculate coverage-metrics such as the ratio of executed data over all possible data.

### 3.1.5 Variants of Coverage

Code coverage data can be aggregated on different levels resulting in different code coverage variants. In our work, we focus on line coverage, therefore we only briefly introduce other variants. We use the source code example Listing 3.1 to illustrate the differences. For this purpose,  $S_{\text{code}} = C_a$  indicates the set of all coverable items,  $C_h$  the set of covered items (“hit”), and  $C_m$  the set of uncovered items (“missed”).

```

1 class Object {
2     public:
3         int i = 1;
4 };
5
6 int f1(int a, double b) {
7     if (a < b && a <= 0) {
8         return a+1;
9     }
10    if (a < b) {
11        return a;
12    }
13    return 0;
14 }
15
16 int f2() {
17     return -1;
18 }
19
20 int f3() {
21     Object object;
22     int result = 2*object.i;
23     return result;
24 }
25
26 int main() {
27     int result1a = f1(1,4);
28     int result1b = f1(1,8);
29     int result1c = f1(1,0);
30     int result3 = f3();
31 }

```

Listing 3.1: Source code example to illustrate coverage variants.

*Function coverage* contains for each function in a program information about its execution. For Listing 3.1, the function coverage is

Function coverage

$$C_a = \{f1, f2, f3, \text{main}\}, \quad (3.4)$$

$$C_h = \{f1, f3, \text{main}\}, \quad (3.5)$$

$$C_m = \{f2\}. \quad (3.6)$$

Note that the class `Object` has a default constructor implicitly generated by the compiler. This implicitly generated constructor may be considered as a function depending on the implementation of a coverage tool.

*Line coverage* contains for each line in each source code file information about its execution. Lines without executable source code are not considered.

Line coverage

For Listing 3.1, the line coverage is

$$C_a = \{1, 6, 7, 8, 10, 11, 13, 16, 17, 20, 21, 22, 23, 24, 26, 27, 28, 29, 30\}, \quad (3.7)$$

$$C_h = \{1, 6, 7, 10, 11, 13, 20, 21, 22, 23, 24, 26, 27, 28, 29, 30\}, \quad (3.8)$$

$$C_m = \{8, 16, 17\}. \quad (3.9)$$

Note that line 24 contains an implicit call to the destructor of object and is therefore an executable line. In total, there are 19 executable lines and 16 lines are covered, therefore the coverage ratio is 0.84. Our definition of line coverage only uses a boolean counter for each line. However, it is also possible to use an integer as a counter for each line that represents the number of times a line was executed during the execution (*execution count*). For Listing 3.1, the execution count for line 16 is 0, 21 is 1, and line 11 is 2.

*Branch coverage* contains for each branch in a program information about its execution. Branches are created by branch instructions and represent different possibilities where the execution continues. In a typical programming language, branches are indicated by *if* statements. However, branches also follow other statements such as a loop, switch or exception statement. The corresponding instruction is typically a (conditional) jump instruction [132]. For Listing 3.1, the branch coverage is

Branch coverage

$$C_a = \{7_1, 7_2, 10_1, 10_2\}, \quad (3.10)$$

$$C_h = \{7_2, 10_1, 10_2\}, \quad (3.11)$$

$$C_m = \{7_1\}, \quad (3.12)$$

where the subscript indicates one of the multiple branches in a line.

*Entry and exit coverage* contains for each entry and exit point of a code unit information about the execution of the entry and exit points. For Listing 3.1, the entry and exit coverage is

$$C_a = \{6, 8, 11, 13, 16, 17, 20, 23, 26, 31\}, \quad (3.13)$$

$$C_h = \{6, 11, 13, 20, 23, 26, 31\}, \quad (3.14)$$

$$C_m = \{8, 16, 17\}. \quad (3.15)$$

All previous coverage variants are based on instruction coverage. However, the following variants require data coverage.

*Condition coverage* contains for each boolean expression information about its evaluation result. For Listing 3.1, we indicate for each boolean expression the evaluation result *true* with the subscript *t*, and the evaluation result *false* with the subscript *f*. Then, the condition coverage is

Condition coverage

$$C_a = \{7_{(a<b)_t}, 7_{(a<b)_f}, 7_{(a<=0)_t}, 7_{(a<=0)_f}, 10_{(a<b)_t}, 10_{(a<b)_f}\}, \quad (3.16)$$

$$C_h = \{7_{(a<b)_t}, 7_{(a<b)_f}, 7_{(a<=0)_f}, 10_{(a<b)_t}, 10_{(a<b)_f}\}, \quad (3.17)$$

$$C_m = \{7_{(a<=0)_t}\}. \quad (3.18)$$

*Decision coverage* contains for each composition of conditions information about its evaluation result. Each decision consists of one or more conditions combined with boolean operators. Decision coverage can also contain information for every entry and exit point whether it was executed or not.

Decision coverage

For Listing 3.1, we only investigate decisions and use the same subscript notation as for condition coverage. Then, the decision coverage is

$$C_a = \{7_{(a<b \ \&\& \ a \leq 0)_t}, 7_{(a<b \ \&\& \ a \leq 0)_f}, 10_{(a<b)_t}, 10_{(a<b)_f}\}, \quad (3.19)$$

$$C_h = \{7_{(a<b \ \&\& \ a \leq 0)_f}, 10_{(a<b)_t}, 10_{(a<b)_f}\}, \quad (3.20)$$

$$C_m = \{7_{(a<b \ \&\& \ a \leq 0)_t}\}. \quad (3.21)$$

*Modified condition/decision coverage* (MC/DC or MCDC) [45, 46, 145] is a rather complex variant of coverage. Therefore, we quote the definition of previous work verbatim [45]:

Modified condition/decision coverage

MCDC is defined as

- every statement in the program has been invoked at least once.
- every point of entry and exit in the program has been invoked at least once.
- every control statement (i.e., branchpoint) in the program has taken all possible outcomes (i.e., branches) at least once.
- every nonconstant boolean expression in the program has evaluated to both a true and a false result.
- every nonconstant condition in a boolean expression in the program has evaluated to both a true and a false result.
- every nonconstant condition in a boolean expression in the program has been shown to independently affect that expression's outcome.

The calculation of MC/DC coverage is rather comprehensive for our example. Therefore, we refer to Section 4.4 for a practical example.

*Parameter value coverage* contains for each parameter of each function information about the different values encountered for this parameter during execution. For Listing 3.1,  $C_a$  and  $C_h$  both contain a large amount of items and the items depend on the definition of *int* and *double*. Furthermore, the *main* function in line 26 has implicit arguments that point to char arrays. Therefore, the size of the set of values for the arguments of *main* is infinite. Given these points, we only list the observed arguments  $C_h = \{6_{a=1}, 6_{b=4}, 6_{b=8}, 6_{b=0}\}$ . Note that the implicit constructor of `Object` also has an implicit parameter `this` that points to the memory location for an object instance. However, listing pointer values is of little use.

Parameter value coverage

### 3.2 Implementation Details

We discuss two implementation details. First, we present techniques to measure coverage. We discuss the idea of counters and their dynamic version. We also highlight zero-overhead techniques and techniques based on functionality provided by the CPU. Second in this section, we present a common format for coverage data that we also use for our work. This thesis does not focus on the implementation of coverage measuring approaches. Therefore, we only provide a general overview of these techniques.

In general, code coverage can be inferred from instruction coverage. The translation requires a mapping between instructions and source code. Compilers such as GCC [238] or Clang [171] provide this mapping. Therefore, we only focus on instruction coverage and data coverage.

### 3.2.1 Counters

We add a counter for each instruction, typically done automatically by a library. More precisely, for each instruction  $i$  in  $S_{\text{instr}}$ , we insert an additional item  $i_c$  before  $i$  that represents a *counter*, i.e., an instruction that modifies the counting variable  $c_i$ . We execute  $P$  and collect the values of all counting variables at the end of the program execution  $PE$ . The output is the instruction coverage  $IC_{PE}$ . We can adapt this approach for data coverage by storing values each time the variable is accessed (read or write).

Counters have multiple advantages. The collection process is simple to understand and can, theoretically, adapt to any sequence of instructions. We would also expect that the results are accurate. However, this approach also has severe disadvantages. The runtime overhead is remarkable because the amount of executions is at least doubled. In practice, the runtime overhead will further increase because optimization techniques employed by the compiler and the CPU are less effective due to the additional statements and increased resource usage. The runtime overhead may be several magnitudes and the resulting total execution time can prohibit the implementation of this approach for large programs. In addition, the practical implementation of this approach faces several challenges such as correct handling of jump instructions or object interactions where non-linear code execution must respect the correct execution of the counting instructions.

Counters are a simple approach.

Counters can have severe performance drawbacks.

### 3.2.2 Dynamic Counters

The execution of counting instructions results in a large runtime overhead. Therefore, the reduction of these executions would reduce the overhead. For this purpose, we can adapt the execution of counting instructions in such a way that they will be removed after their execution (or replaced with an empty “*NOP*” instruction). With such an approach, the overhead would only occur for the first execution of a counting instruction but not for subsequent executions. This can be implemented by self-modifying code or by a virtual machine that modifies the code-to-be-executed in such a way that at the first time of the execution, the counting instruction is added to the sequence of instructions and for further executions, the original sequence of instructions is restored [34, 67, 101, 248].

Dynamically modify binary code.

Compared to permanent counters, the advantage of this approach is a possible reduction of the overhead factor due to the following effects:

- The compiler can leverage optimization techniques because no additional counting instructions must be considered.
- The amount of executed counting instructions may be smaller because the instructions will be removed after their execution. This results in fewer instructions executed but also in a more effective execution due to optimization techniques applied by the CPU.

However, the process of dynamic modification may introduce additional overhead. In addition, it is only suitable when we only require information on whether an instruction was executed or not. For SAP HANA, the tool *DynamoRIO drcov* [67] collects such coverage information.

### 3.2.3 Emission Observation

A CPU executing instructions emits electromagnetic emanations. It is possible to learn patterns in these emanations in a training phase. Then, in a profiling phase, we can reveal the executed instructions based on the learned patterns [38, 223]. This allows coverage measurements with zero overhead to the program execution. However, the related work is in an early stage and shows limited reliability with about 90% path prediction accuracy [223] for single core executions. Multi-core execution reduces the accuracy.

Identify CPU operations by previously learned emission patterns.

### 3.2.4 Functionality Provided by CPU

Current CPU architectures provide specific functionality to measure coverage with a low overhead compared to software approaches. Intel provides Last Branch Record (LBR) and Processor Trace (PT) for selected CPU models [131, 214, 229].

A CPU that supports LBR records the last 4 to 32 branch results in a set of model-specific register (MSR) as a tuple of “from” and “to” addresses. The MSR represent a ring-buffer, i.e., the last recent entries overwrite the oldest results if the buffer is full. An additional register points to the last recent entry in the ring-buffer. Intel argues that the LBR has no practical overhead. However, due to the limited size, execution must be halted frequently to read the content of the MSR. These halts then add considerable overhead. Therefore, the LBR is typically only used for sampling, i.e., the content is read in fixed time intervals assuming that statistically the result is representative for the program execution.

LBR stores the result for up to 32 branches.

A CPU that supports PT records the program execution in a compressed format in memory. Compared to LBR, this has several advantages. First, PT is not limited to the last  $n$  branches. Second, data is written to the memory and the CPU must not halt to allow data access. Third, PT allows investigating a larger time frame compared to uncompressed LBR data. However, there are also limitations. First, PT impacts the execution time. Depending on the memory load of an application, the runtime overhead can be a factor of 1.05 to 1.20 [229] or, for multithreaded applications, up to a factor 20 [244]. Second, a typical CPU executes about  $3 \times 10^9$  instructions per second, assuming 20% branch instruction [4], we generate  $6 \times 10^8$  bits per second. Intel reports 100 MB/s of compressed data and a compression ratio of 10. Therefore, the amount and bandwidth of memory and disk can limit the usage scenario for PT to short term measurements.

Intel processor trace for full execution details.

Small overhead.

### 3.2.5 Format for Code Coverage

For this work, we use the LCOV [15] format to store coverage information in the filesystem. Fig. 3.4 provides an example for line coverage data where *SF*: indicates a source file, *DA*: indicates the data for the current source file, i.e., the line number and whether it is covered (“hit”, indicated by a number larger than 0) or not (indicated by the number 0) and *end\_of\_record* indicates the end of a source file. The optional *LF*: summarizes lines found and *LH*: indicates the lines hit. In the given example, the single file *file.c* has 6 executable lines and 3 of them were executed (“covered”).

```

1 SF:/path/example.c
2 DA:13,0
3 DA:14,1
4 DA:15,1
5 DA:46,0
6 DA:47,0
7 DA:48,1
8 LF:6
9 LH:3
10 end_of_record

```

Figure 3.4: Example of the LCOV data format for a single source file.



While working with line coverage data, we use *bitsets*. For a source file  $SF$  with  $m$  lines, we use a set with  $m$  bits where bit  $i$  indicates whether line  $i$  in  $SF$  was executed or not by a specific execution of  $P$ . Technically, the set is implemented by 64 bit numbers and we use bit operations to set or read a specific bit as shown by Fig. 3.5. More sophisticated implementations are available, such as roaring bitmaps that utilize a compression algorithm [44, 174]. Furthermore, the set of all coverable lines is typically considerably larger compared to the set of covered lines by a single test. Therefore, we utilize a sparse set, where we allocate a block of memory for bits only if at least one of the bits within this block is set.

Bitsets

```

1 public class BitSet64 {
2     static final int LOWER_BITS_MASK = 0b111_111; //63=64-1
3     static final int LOWER_BITS_SHIFT_SIZE = 6; //2^6=64
4     private final long[] array;
5     public BitSet64(final long max) {
6         if (max >= 137438952896L) // (2^32-1-8) * 64
7             throw new IllegalArgumentException("max>=137438952896");
8         array = new long[(int)(max >> 6)+1]; //+1 index to length
9     }
10    public void set(final long i) {
11        array[(int) (i >> LOWER_BITS_SHIFT_SIZE)]
12            |= (1L << (i & LOWER_BITS_MASK));
13    }
14    public void clear(final long i) {
15        array[(int) (i >> LOWER_BITS_SHIFT_SIZE)]
16            &= ~(1L << (i & LOWER_BITS_MASK));
17    }
18    public void flip(final long i) {
19        array[(int) (i >> LOWER_BITS_SHIFT_SIZE)]
20            ^= (1L << (i & LOWER_BITS_MASK));
21    }
22    public boolean get(final long i) {
23        return (array[(int) (i >> LOWER_BITS_SHIFT_SIZE)]
24            & (1L << (i & LOWER_BITS_MASK))) != 0;
25    }
26 }

```

Figure 3.5: A simple bitset implementation in Java to show the essential concept and operations of a bitset.

A bitset data structure has several practical advantages for our work. The required space to store line coverage data is reduced compared to the LCOV format. We evaluate the improvements in Section 5.4. A reduction in space requirements also reduces the execution time due to better cache usage. In addition, the bitset allows us to use block-wise boolean operations which further decrease the execution time. Overall, the bitset data structure facilitates a fast analysis and fast algorithms for line coverage data, even in the case of rather large initial data.

### 3.3 Problems and Algorithms on Coverage Data

We investigate several algorithms that work on coverage data. To simplify the presentation and discussion of the algorithms, we state several assumptions and define mathematical operations on coverage data. We then introduce the set cover problem and several variants. Following the problem definitions, we present and discuss algorithms to solve these problems. In this context, we also present implementations in Java 8 for several algorithms to highlight and discuss practical considerations.

### 3.3.1 Assumptions

We list several assumptions that either are required by the following algorithms or simplify the discussion and analysis of these algorithms. We also discuss whether these assumptions affect generality.

We assume that the coverage data is line coverage, that is, a sequence of integers indicating which lines are covered. However, the algorithms can also support other types of coverage, e.g., function coverage or branch coverage. For example, we can convert line coverage data to function coverage data.

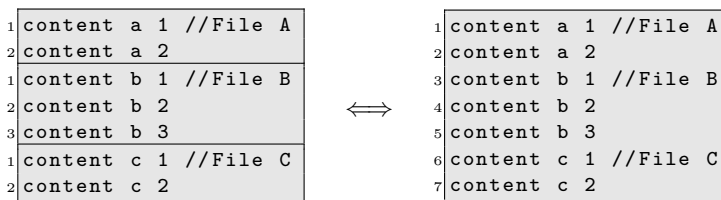
Line coverage.

We assume that for a test run, i.e., the execution of all tests, we obtain  $n$  sets of line coverage. Each of these  $n$  sets represents the coverage data for a single test or for a test suite. The value of  $n$  depends on the grouping of the tests. We achieve a maximal  $n$  if we run each single test case separately and measure the coverage for each execution. Alternatively, we achieve a minimal  $n$  (1, in fact) if we execute all test cases in a single execution and measure the coverage for all of them together. In practice, it is not feasible to measure coverage for each test case separately due to the overhead of the test setup as discussed in Section 2.2.3.2. However, a single coverage measurement of all test executions would only provide a rather limited amount of information as we are unable then to identify which specific tests executed a specific line of source code. Therefore, in practice, test cases are organized in test suites given by a file-based structure and each test suite contains one or several tests. A coverage run then measures coverage information for each of these test suites separately.

A test run consists of multiple test suite executions and therefore multiple coverage files.

We assume that the coverage data originates only from a single source code file. This abstraction simplifies the discussion and analysis of algorithms without impacting the results. In practice, software such as SAP HANA consists of more than 10 000 files. However, we can map multiple files to a single file with the following mapping: for each source file  $SF$ , we rename each line number  $n$  to  $SF-n$ . We concatenate the coverage data of all files to a new source file  $SF_{all}$  that has coverage information for each renamed line. We access the lines in  $SF_{all}$  by a new index from 1 to  $m$ , where  $m$  is the sum of all lines in all files. The following example visualizes the concept:

A single source file.



We filter lines that are not executed by any test (“uncovered lines”) if the result of an algorithm does not depend on such uncovered lines. Note that this does not affect lines that are not executed by some tests but are executed by at least one test. This step does not affect generality.

Filter uncovered lines.

At several places, we provide an example implementations to simplify the presentation and show the practical usage. For this purpose, we use Java and assume a version of at least 10. Additionally, we assume the existence of a `Coverage` class that implements operations listed in Section 3.3.2 and provides access to the coverage data by a `Set<String, Set<Integer>>`

data structure representing filenames with executed lines. Furthermore, the class implements `hashCode` and `equals(Objects)` in a suitable way. Finally, all variables in this class are final, therefore making an instance of this class immutable. We omit to present the implementation of this class because it is rather long and uninteresting.

### 3.3.2 Operations on Code Coverage

We define multiple operations on coverage data, which are, in fact, set operations. The introduction of a different notation for common set operations simplifies the discussion of the following algorithms.

Let  $L$  be a sequence  $\langle 1, \dots, n \rangle$  of all source code line numbers in a software program  $P$  with  $n$  lines. Then, line coverage data  $C$  is a subsequence  $C \subseteq L$  where each item in  $C$  identifies a line executed for a specific execution  $PE$  of  $P$ . The set of all possible coverage data is  $C_{all} = \mathcal{P}(L)$ , i.e., the power set of  $L$ . We define the following operations for  $c, d \in C_{all}$ :

*add/addition/union*:  $c + d := c \cup d$  — we add two coverage data by taking the union of both sets.

add, sub, and, xor on coverage data.

*sub/subtraction/relative complement*:  $c - d := c \setminus d$  — we subtract  $d$  from  $c$  by removing all items in  $c$  that are contained in both coverage data.

*and/intersection*:  $c \& d := c \cap d$  — for two coverage data, we keep only the elements that are contained in both sets.

*xor/symmetric difference*:  $c \oplus d := c \Delta d$  — for two coverage data, we keep only the elements that are contained in exactly one set.

The notation *name1/name2* indicates interchangeable names for the same operation. It follows from set operations that *add*, *and*, and *xor* are commutative operations and *sub* depends on the order of the arguments.

We demonstrate the notations with several examples:

$$\begin{array}{l} c = \{1, 2, 3\}, \\ d = \{3, 5\} \end{array} \implies \begin{array}{l} c + d = \{1, 2, 3, 5\}, \\ c - d = \{1, 2\}, \\ d - c = \{5\}, \\ c \& d = \{3\}, \\ c \oplus d = \{1, 2, 5\}. \end{array}$$

### 3.3.3 Metrics: Distance Functions

Given a sequence of lines  $L = \langle 1, \dots, n \rangle$  for a software program  $P$  and two line coverage data files  $c_1, c_2 \subseteq L$  representing the executed lines for two test executions, we may ask how similar these two coverage files are. In the case  $c_1 = c_2$ , they are obviously very similar, in fact they are equal. In the case when  $c_1$  is inverted to  $c_2$ , i.e., every line that is executed in  $c_1$  is not executed in  $c_2$  and vice versa, we could say the similarity is the lowest, in fact they share not a single line with the same execution state. These corner case are simple to illustrate. However, for all other cases the similarity depends on of how define similarity. Mathematically, we can understand a line coverage file that contains information about the execution of  $n$  lines as a vector in

$\mathbb{B}^n$ , where  $\mathbb{B} = \{0, 1\}$ . We can define a metric (called distance)  $d$  on the described vectorspace [107]:

$$d : \mathbb{B}^n \times \mathbb{B}^n \mapsto [0, \infty[, \quad (3.22)$$

$$(3.23)$$

where

$$d(x, y) \geq 0 \quad (3.24)$$

$$d(x, y) = 0 \Leftrightarrow x = y \quad (3.25)$$

$$d(x, y) = d(y, x) \quad (3.26)$$

$$d(x, y) \leq d(x, z) + d(z, y) \quad (3.27)$$

$$\forall x, y, z \in \mathbb{B}^n. \quad (3.28)$$

We define several metrics to measure distances between coverage files. They serve different purposes as we explain for each definition. We continue to use  $x, y$  for vectors in  $\mathbb{B}^n$  and  $c_1, c_2$  for coverage files. Note that coverage files do not contain lines that were not executed. Therefore, we must virtually extend such a line in  $c_1$  if not contained in  $c_1$ , but in  $c_2$  and vice-versa.

The *Euclidean metric* [11]  $d_e$  measures the distance as a “straight line” between the two points indicated by the input vectors:

Related work also uses distance metrics for fault localization [111].

Euclidean metric

$$d_e(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}. \quad (3.29)$$

Interpreted in the context of coverage, the Euclidean metric counts the (square root of) number of times when the entries in the two coverage files differ for any source code line:

$$d_e(c_1, c_2) = \sqrt{|c_1 \oplus c_2|}. \quad (3.30)$$

The *Unshared metric*  $d_u$  the Euclidean metric without the square root:

Unshared metric

$$d_u(x, y) = \sum_{i=1}^n |x_i - y_i|. \quad (3.31)$$

Interpreted in the context of coverage, the unshared metric counts the number of times the two coverage files differ for a source code line:

$$d_u(c_1, c_2) = |c_1 \oplus c_2|. \quad (3.32)$$

Several previous work utilize the *Proportional Binary metric*  $d_p$  [74, 143, 175]. We quote Leon and Podgurski: “(The proportional binary metric) is a modified Euclidean distance formula that takes into account whether an element was covered or not, and also the count of how many times the element was executed, while adjusting for differences in scale between counts.” [175]. Furthermore, they give the following definitions. First, they introduce the functions  $C, P, B$ :

Proportional Binary metric

$$C_{i,j} = \text{number of times element } j \text{ is executed for test run } t_i \quad (3.33)$$

$$P_{i,j} = \frac{C_{i,j} - \min_k \{C_{k,j}\}}{\max_k \{C_{k,j}\} - \min_k \{C_{k,j}\}} \quad (3.34)$$

$$B_{i,j} = \begin{cases} 0 & \text{if } P_{i,j} = 0, \\ 1 & \text{if } P_{i,j} \neq 0. \end{cases} \quad (3.35)$$

Then, the proportional binary metric is:

$$d_p(x, y) = \sqrt{\sum_k (P_{x,k} - P_{y,k})^2 + |B_{x,k} - B_{y,k}|}. \quad (3.36)$$

We adapted the metric name  $d_p(x, y)$  to our naming schema, the original definition uses  $D_{n,m}$ . Interestingly,  $k$  is not further defined in several previous work. Dickinson et al. do not provide any mathematical definition of the metric at all [74]. Leon and Podgurski do not further define  $k$  [175]. Jiang et al. state that  $k$  is the number of test cases in total [143]. However, we can deduce that the  $k$  in the definition for  $d_p(x, y)$  (Eq. (3.36)) is different to the  $k$  in the definition of  $P$  (Eq. (3.34)). The former seems to be the number of coverable elements, the latter the number of tests.

Inconsistencies in definitions.

We can show that the  $k$  are different by type analysis on the definitions. For  $C_{i,j}$ , the first index is of type test index, the second of type coverable element. As  $P_{i,j}$  uses  $C$  (Eq. (3.34)), we can conclude that the first index of  $P$  must be of type test index, the second of type coverable element. Thus, we can conclude that in the definition of  $d_p(x, y)$  (Eq. (3.36)), the index  $k$  of the sum, which is used as a second index to  $P$  (and  $B$ ) must be of type coverable element. It also follows that  $k$  cannot be the number of tests in the definition of  $d_p(x, y)$ . Given these points, we adapt the definition to clarify the ambiguity. First, we define the sequence  $CE$  of all coverable elements (such as lines for line coverage) and the set of all coverage files  $CF$ . Then,

$$d_p(x, y) = \sqrt{\sum_{i=1}^n (P_{x,i} - P_{y,i})^2 + |B_{x,i} - B_{y,i}|}, \quad (3.37)$$

where

$$n = |CE| \text{ (the number of all coverable elements)} \quad (3.38)$$

$$k = |CF| \text{ (the number of all coverage files)} \quad (3.39)$$

$$P_{i,j} = \frac{C_{i,j} - \min_k \{C_{k,j}\}}{\max_k \{C_{k,j}\} - \min_k \{C_{k,j}\}} \quad (3.40)$$

$$C_{i,j} = \text{number of times statement } j \text{ is executed for test } t_i \quad (3.41)$$

$$\min_k \{C_{k,j}\} = \min\{C_{1,j}, \dots, C_{k,j}\} \quad (3.42)$$

$$\max_k \{C_{k,j}\} = \max\{C_{1,j}, \dots, C_{k,j}\} \quad (3.43)$$

$$B_{i,j} = \begin{cases} 0 & \text{if } P_{i,j} = 0, \\ 1 & \text{if } P_{i,j} \neq 0. \end{cases} \quad (3.44)$$

Note that the denominator in Eq. (3.40) can be 0 if a specific coverable element has the same state for all test executions. Previous work does not address this issue. We assume that in such cases, the corresponding element is not considered or  $P_{i,j} = 0$ .

In our analysis, we focus on line coverage that only indicates whether a line is executed at all (and not the count). There,

$$P_{i,j} = \frac{C_{i,j} - \min_k \{C_{k,j}\}}{\max_k \{C_{k,j}\} - \min_k \{C_{k,j}\}} = \frac{C_{i,j} - 0}{1 - 0} = C_{i,j}. \quad (3.45)$$

This follows because  $\min_k\{C_{k,j}\}$  must be 0, otherwise the denominator would be 0. Conclusively,  $\max_k\{C_{k,j}\}$  must be 1 to avoid 0 as denominator. Based on Eq. (3.45) and Eq. (3.44), it follows that

$$B_{i,j} = C_{i,j}, \quad (3.46)$$

and therefore we can simplify Eq. (3.37) to

$$d_p(x, y) = \sqrt{\sum_{i=1}^n (C_{x,i} - C_{y,i})^2 + |C_{x,i} - C_{y,i}|} = \sqrt{\sum_{i=1}^n 2 \times (x_i - y_i)^2},$$

because  $|x - y| = (x - y)^2$  for  $x, y \in 0, 1$ . Interpreted in the context of line coverage, the proportional binary metric is similar to the Euclidean metric, but uses a factor of  $\sqrt{2}$  as a scaling difference:

Proportional binary metric is similar to Euclidean metric.

$$d_p(c_1, c_2) = \sqrt{2 \times |c_1 \oplus c_2|}. \quad (3.47)$$

Furthermore, we define a *Shared function*  $f_s$  as counting the number of times the  $i$ -th index of each vector equals to one for both vectors:

Shared function

$$f_s(x, y) = \sum_{i=1}^n x_i \times y_i. \quad (3.48)$$

Interpreted in the context of coverage, the shared metric counts the number of lines that are marked as executed in both coverage files:

$$f_s(c_1, c_2) = |c_1 \& c_2|. \quad (3.49)$$

We can verify that  $f_s$  is not a metric, but is symmetric:

$$c_1 = \{1\} \implies f_m(c_1, c_1) = 1 \neq 0, \text{ and} \quad (3.50)$$

$$\forall x, y \in \mathbb{B}^n : f_s(x, y) = \sum_{i=1}^n x_i \cdot y_i = \sum_{i=1}^n y_i \cdot x_i = f_s(y, x). \quad (3.51)$$

Therefore,  $f_s$  still allows us to compare the similarity of two coverage files as a similarity measure, but not in the sense of a typical distance.

In addition to the shared function, we also define a *Sub Function*  $f_m$  that counts the number of times the first index is 1 but the second is 0:

Sub Function

$$f_m(x, y) = \sum_{i=1}^n x_i \times (x_i - y_i), \quad (3.52)$$

where  $m$  can be read as “minus”. Interpreted in the context of line coverage, the sub function subtracts the second coverage from the first coverage and counts the number of executed lines in the result:

$$f_m(c_1, c_2) = |c_1 - c_2|. \quad (3.53)$$

The function  $f_m$  is useful to compare coverage data but is not a metric. For instance, the symmetry (Eq. (3.26)) is not fulfilled:

$$c_1 = \{1\}, c_2 = \{\} \implies f_m(c_1, c_2) = 1 \neq 0 = f_m(c_2, c_1). \quad (3.54)$$

To conclude, we show examples for all metrics:

$$\begin{aligned} c_1 = \{1, 3, 5, 6\} &\implies x = (1, 0, 1, 0, 1, 1)^t \in \mathbb{B}^6. \\ c_2 = \{1, 2\} &\implies y = (1, 1, 0, 0, 0, 0)^t \in \mathbb{B}^6. \end{aligned}$$

Then,

$$\begin{aligned} d_e(x, y) &= 2, & f_s(x, y) &= 1, & d_u(x, y) &= 4, \\ d_p(x, y) &= \sqrt{2} \times 2, & f_m(x, y) &= 3, & f_m(y, x) &= 1. \end{aligned}$$

Note that we include line 4 in these examples, which is not executed in any test execution related to coverage data  $c_1, c_2$ . As we can see, such lines do not affect any metric. Therefore, we typically ignore lines that are not executed by any test and exclude them in any analysis.

### 3.3.4 Set Cover Problems

We present the set cover problem and a variant of the set cover problem, the weighted set cover problem. We first introduce the *set cover problem* (SCP). Given a universe  $U$  of  $n \in \mathbb{N}_0$  elements and a family  $F$  of  $m \in \mathbb{N}_0$  subsets of  $U$  so that the union of all subsets is the universe:

Set cover problem

$$U = \{e_1, e_2, \dots, e_n\} \tag{3.55}$$

$$F = \{S_1, S_2, \dots, S_m\} \tag{3.56}$$

$$S_i \subseteq U \text{ and } \bigcup_{S \in F} S = U. \tag{3.57}$$

The SCP asks to find a subset of  $F$  whose union equals the universe and the cardinality is minimal:

$$\text{Find } R \subseteq \{1, 2, \dots, m\} \tag{3.58}$$

$$\text{such that } \bigcup_{i \in R} S_i = U \tag{3.59}$$

$$\text{and } |R| \leq |X| \tag{3.60}$$

$$\forall X \subseteq \{1, 2, \dots, m\} \text{ where } \bigcup_{i \in X} S_i = U. \tag{3.61}$$

The set cover problem (and its variants) is intensively studied [251]. Strictly speaking, we differentiate between the decision version and the optimization version of the SCP. The decision version asks whether there exists a solution with a given size, i.e., a fixed amount of sets. The optimization problem asks to find a solution with the smallest size. The decision version is NP-complete and is also one of the classical 21 NP-complete problems [148]. The optimization version is NP-hard [148, 162, 251]. We are only interested in the optimization version. Therefore, from now on, SCP refers to the optimization version as given by the definition above.

Decision version and optimization version

The set cover problem is NP-complete.

We can relate the SCP to line coverage by a short transformation. Our universe  $U$  is a set of line numbers and the family of subsets  $F$  represents test executions where each set contains line numbers executed by a test. Given these  $m$  test executions, we can then ask to find the minimal amount of test executions to cover all lines in  $U$  (the union of all test executions).

SCP and line coverage

Practically, we ask for the minimal amount of tests to cover all (coverable) lines. Note that such reductions may not be safe in terms of bug-finding ability. Tests that execute the same lines may show different behavior.

The *weighted set cover problem* (WSCP) is a modification to the SCP. In addition to the definition of the SCP, we attribute weights (or “costs”) to each subset of  $U$  in  $F$ . Let  $\mathbb{W} : F \mapsto \mathbb{N}^+$  be a function that attributes a weight  $w_i$  to each  $S_i \in F$ , then the WSCP asks to find a subset of  $F$  whose union equals the universe and the sum of all associated weights is minimal:

$$\text{Find } R \subseteq \{1, 2, \dots, m\} \quad (3.62)$$

$$\text{such that } \bigcup_{i \in R} S_i = U \quad (3.63)$$

$$\text{and } \sum_{i \in R} w_i \leq \sum_{i \in X} w_i \quad (3.64)$$

$$\forall X \subseteq \{1, 2, \dots, m\} \text{ where } \bigcup_{i \in X} S_i = U. \quad (3.65)$$

Weighted set cover problem

We can relate the WSCP to line coverage and tests execution time by using the same transformation as for the SCP before and additionally associate weights in terms of execution time (say, in minutes) to each test and therefore to each set of covered lines.

WSCP: line coverage and execution time

The SCP and the WSCP are NP-hard. Therefore, we do not expect to find an algorithm with a polynomial time complexity that guarantees to find an optimal solution for all inputs. For example, simply enumerating all combinations is not feasible in practice. For SAP HANA, the problem size typically consists of 4 million elements (lines) within 2000 subsets. Enumerating all solutions is equivalent to setting bits in a binary number with 2000 bits. Each bit represents whether a specific subset is included or not. Therefore, enumerating all combinations would require calculating  $2^{2000} \approx 10^{602}$  possible inputs. Furthermore, for each input, we must iterate over 4 million elements. Given these points, it is not feasible to enumerate all combinations for this problem size. However, even for large problem instances, we can apply an approach consisting of the following steps:

SCP and WSCP are NP-hard

- Apply logical reductions with polynomial time complexity to potentially simplify a given problem instance, i.e., reduce the cardinality of  $F$  and the cardinalities of the elements in  $F$ .
- Find an optimal solution by a full search for small problem sizes.
- Find a near-optimal solution with a heuristic, i.e., a polynomial time greedy algorithm that achieves an approximation ratio of  $H(n)$  where  $n$  is the cardinality of  $U$ . In this case,  $H(n)$  is the  $n$ -th harmonic number, i.e.,  $H(n) = \sum_{i=1}^n 1/i \leq \ln(n) + 1$ .

We discuss these steps in the following sections. There is also a wide range of research proposing different heuristics [30, 79, 161].

### 3.3.5 Logical Reductions for the SCP

We can apply several logical reductions for the SCP in a repeated approach, namely *duplicate removal*, *subset removal* and *must-have item identification*. Each of these reductions has an execution time that depends on the current



problem size by a polynomial function. We also see that the effort to parallelize these reductions is considerably low, therefore further reducing the overall execution time. For the discussion of the worst-case algorithmic time complexity, we assume that  $m = |F|$  dominates the execution time. In this case,  $m$  is the amount of executed test suites and therefore the amount of coverage data files. To simplify the presentation, we ignore  $n = |U|$ . While  $m$  may vary for a test run, the source code  $U$  does not.

For *duplicate removal*, we identify the sequence  $S_d$  of all elements in  $F$  that have identical elements:

$$S_d = \langle S_i \mid \exists S_j \in F : S_i = S_j \text{ and } i < j \rangle. \quad (3.66)$$

Note that formally, sets do not contain duplicates. We either assume that  $F$  is a multiset or that each item in  $F$  additionally has a label representing the corresponding coverage file with operations adapted correspondingly. We omit these details in favor of the brevity of the presentation.

In the duplicate removal step, we identify all tests that execute the same set of lines. We then keep only a single representative of each group of duplicates. This may seem ineffective because system tests, as explained in Section 5.6.1, typically contain random coverage that prevents duplicate detection. However, due to the repeated application of all logical reductions, the duplicate detection step is quite effective after the *must-have item identification* (see later), which filters such random coverage. The implementation of duplicate detection is rather short and trivial, as shown by Fig. 3.6. The algorithm can be trivially parallelized by using `parallelStream`, but typically the execution time is less than a few seconds due to the efficient `hashCode` utilization within the `HashSet`. `HashSet` provides (amortized) linear time for adding elements, therefore the overall time is linear with  $\mathcal{O}(m)$  where  $m = |F|$ , i.e., the number of test executions with coverage.

Duplicate removal

Amortized linear time complexity

```

1 public List<Coverage> removeDuplicates(List<Coverage> cov) {
2     // we assume that:
3     // 1) Coverage implements hashCode() and equals(Object)
4     // 2) Collectors.toSet() uses a HashSet
5     return cov.stream().collect(Collectors.toSet());
6 }

```

Figure 3.6: Identification and removal of coverage duplicates.

The *subset removal* extends the *duplicate removal*. We identify the set  $S_u$  of all elements that are subsets to any element in  $F$ , more formally:

$$S_u = \langle S_i \mid \exists S_j \in F : S_i \subseteq S_j \text{ and } i < j \rangle. \quad (3.67)$$

Practically, we remove all test executions where the covered lines are also executed by any other test execution. Subset removal includes duplicate removal, because a set  $X_i$  that is equal to set  $X_j$  is also a subset in our definition. The reason why we distinguish between subset removal and duplicate removal is the algorithmic time complexity. Our implementation for duplicate removal has a linear worst-case algorithmic time complexity. However, depending on the implementation, the subset removal has a time complexity  $\mathcal{O}(m^2)$  where  $m = |F|$ . It remains unclear whether an algorithm with linear time exists for this problem.

Subset removal

Quadratic time complexity

A naive implementation for subset removal with time complexity  $\mathcal{O}(m^2)$  could utilize a matrix-based comparison where we compare each element in  $F$  with all other elements in  $F$ . Fig. 3.7 shows an implementation that improves the naive matrix-based comparisons. Instead of comparing all tuples, we only calculate comparisons for an upper triangle. For this purpose, we sort the input by cardinality and then exploit the property that for two sets  $S_1, S_2$ ,  $|S_1| > |S_2| \implies S_1 \not\subseteq S_2$ , because there must be at least one element in  $S_1$  that is not in  $S_2$ . Then, for a list of sets ordered by cardinality, we can only find a superset for the set at index  $i$  at an index  $j > i$ . We cache the cardinality calculation and calculate the subset detection in parallel

This implementation still has a computational complexity of  $\mathcal{O}(m^2)$ , because the amount of operations depends on the number of elements in the upper triangle (without the diagonal line) which is  $(m^2 - m)/2$ . However, in practice, it requires less than half of the work compared to the naive matrix-based comparison. We only visit the upper triangle and we can abort the check if we found at least one superset. The implementation shown in Fig. 3.7 typically executes within less than 10s for the problem sizes encountered at SAP HANA.

```

1 public List<Coverage> removeSubsets(final List<Coverage> cov){
2     final List<Coverage> all = new ArrayList<>(cov);
3     // calculating the cardinality might be expensive, cache it
4     Map<Coverage, Integer> mapCovToCardinality = all.stream().
5         collect(toMap(i -> i, i -> i.getCardinality()));
6     sort(all, Comparator.comparingInt(mapCovToCardinality::get));
7     final IntPredicate isNoSubSet = i -> isNoSubSet(all.get(i),
8         all.subList(i + 1, all.size()));
9     Set<Coverage> ok = IntStream.range(0, all.size()).parallel()
10        .filter(isNoSubSet).mapToObj(all::get).collect(toSet());
11    // keep original order
12    return cov.stream().filter(ok::contains).collect(toList());
13}
14
15 public boolean isNoSubSet(Coverage set, List<Coverage> all) {
16     for (final Coverage other : all) {
17         if (set.isSubSetOf(other))
18             return false;
19     }
20     return true;
21}

```

Figure 3.7: Identification and removal of coverage subsets.

For *must-have item identification*, we exploit Eq. (3.59) that the solution must contain all elements. For any element  $e$  in our universe  $U$  that exists only in a single subset  $S_e$  of  $F$ , we can conclude that  $S_e \in R$ , i.e., the set  $S_e$  must be included in the solution because it is the only option to include  $e$ . Therefore, we can collect all sets of  $F$  that contain such required elements into a partial solution  $S_p$  and reduce the problem size to  $F \setminus S_p$ . Furthermore, we know that  $S_p \subseteq R$ . Therefore, we can conclude that not only the original elements of interest, but all elements in  $\bigcup_{S \in S_p} S$  are already covered for a solution. Thus, we can remove these elements from all sets in  $F \setminus S_p$  and thereby we can further reduce the problem size.

Must-have item identification

The implementation shown in Fig. 3.8 and Fig. 3.9 identifies all sets in  $F$  that contain an element that appears only once in all sets. Practically, the sets are tests and the elements covered lines. The implementation is separated into several parts to facilitate parallelization:

1. Calculate the universe  $U$ , i.e., the set of all lines.
2. Create data structures for later use.
3. Use the data structures of the previous step to count the occurrence of each line in all coverage data files.
4. Identify all coverage files that contain a line with a count of 1.

As all steps depend on the number of tests, the algorithmic time complexity is  $\mathcal{O}(m)$ , where  $m = |F|$ . All steps, expect the data structure creation, use a parallel implementation. The creation of the data structures is faster single threaded due to the synchronization overhead for parallelization.

Linear time complexity

```

1 /** Find all tests with lines that appear only once.*/
2 public Set<Coverage> getMustHaveItems(List<Coverage> cov) {
3   // plan: create map file->line->#tests, filter for #tests=1
4   final Map<String, Map<Integer, AtomicInteger>>
5     mapSourceToTestCountPerLine = new HashMap<>();
6   // 1) get the sum to iterate over everything
7   final Coverage sum = parallelSum(cov);
8   // 2) we create our data structures
9   sum.getMapSourceToLhit().entrySet().stream().forEach(entry
10    -> mapSourceToTestCountPerLine.put(entry.getKey(),
11    convertLinesToMapAtomicIntegers(entry.getValue())));
12 // 3) now we count in parallel
13 cov.parallelStream().forEach(i->i.getMapSourceToLhit().
14   entrySet().stream().forEach(e->e.getValue().stream().
15   forEach(l->mapSourceToTestCountPerLine.get(e.getKey()).
16   get(l).incrementAndGet())));
17 // 4) identification
18 final Set<Coverage> result = ConcurrentHashMap.newKeySet();
19 mapSource2TestCountPerLine.entrySet().parallelStream().
20   forEach(e -> find1Hit(e, cov, result));
21 return result;
22 }

```

Figure 3.8: Identification of must-have items for the set cover problem, parallelized.

```

1 public static Map<Integer, AtomicInteger>
2   convertLinesToMapAtomicIntegers(Set<Integer> lines) {
3   return lines.stream().collect(toMap(i -> i, i -> new
4   AtomicInteger()));
5 }
6 public static void find1Hit(Entry<String, Map<Integer,
7   AtomicInteger>> sourceToTestCountPerLine, List<Coverage>
8   coverage, Set<Coverage> result) {
9   final String source = sourceToTestCountPerLine.getKey();
10  final Map<Integer, AtomicInteger> mapLineToTestCount =
11    sourceToTestCountPerLine.getValue();
12  for (final Entry<Integer, AtomicInteger> entryLineToCount :
13    mapLineToTestCount.entrySet()) {
14    if (entryLineToCount.getValue().get() != 1)
15      continue;
16    final Integer line = entryLineToCount.getKey();
17    for (final Coverage item : coverage) {
18      if (item.isSourceLineHit(source, line)) {
19        result.add(item);
20        break; // there can be only one item
21      }
22    }
23  }
24 }

```

Figure 3.9: Utility methods for must-have item identification.

After the identification of must-have items, we reduce the problem as shown by Fig. 3.10. We add all found items to the solution  $R$ . Then, we

remove  $R$  from  $F$  and delete all elements in the union of  $R$  from the universe  $U$ . Finally, we remove all items in  $U$  that are already in the solution  $R$ . The algorithmic time complexity for the identification of must-have items remains to be  $\mathcal{O}(m)$ , where  $m = |F|$ .

```

1 public List<Coverage> removeMustHaveItems(List<Coverage>
   coverage, Set<Coverage> solution) {
2   Set<Coverage> mustHave = getMustHaveItems(coverage);
3   if (mustHave.isEmpty()) // no reduction found
4     return coverage;
5   solution.addAll(mustHave);
6   final Coverage sum = parallelSum(mustHave);
7   Predicate<Coverage> notSelected = i->!mustHave.contains(i);
8   final Function<Coverage, Coverage> subSum = i->i.sub(sum);
9   final Predicate<Coverage> removeEmpty = i -> !i.isEmpty();
10  return coverage.parallelStream().filter(notSelected).map(
   subSum).filter(removeEmpty).collect(toList());
11 }

```

Figure 3.10: Removal of must-have items for the set cover problem.

Finally, the *repeated approach* combines all previous reductions as shown in Fig. 3.11. The combination of *duplicate/subset removal* and *must-have item identification* and their repeated application results in an interesting effect. Line coverage data is typically continuous. Therefore, *subset removal* is effective in removing such chunks. However, a typical limitation case of *subset removal* are lines contained only in a single set, because such sets can never be a subset of any other set. For such cases, the *must-have item identification* is effective because all such instances will be removed. Similarly, a typical limitation case of *must-have item identification* are elements that are contained in a large number of sets. For this case again, the *subset removal* is effective. Overall, each approach reduces the problem in such a way that another approach can find new reductions.

Repeated approach

After we finished the repeated approach phase, i.e., when find no further reductions, we solve the remaining problem. For up to 15 elements, we enumerate all possible solutions (up to  $2^{15}$  combinations), which typically requires less than 1 min. In such a case, we can conclude that the solution is optimal. The optimality is guaranteed by the properties of the logical reductions and the full enumeration of all remaining cases. For problem sizes with more than 15 elements, we apply a greedy approach (see later). In this case, we are unable to make any statement about the optimality.

Repeated application of logical reductions efficiently reduces the problem size

We use the following example to demonstrate the algorithm:

$$\begin{aligned}
 U &= \{1, 2, 3, 4, 5, 6, 7\} \\
 F &= \{S_1, S_2, S_3, S_4, S_5, S_6, S_7\} \\
 S_1 &= \{1, 2, 3, 4\} \\
 S_2 &= \{1, 3, 4, 5\} \\
 S_3 &= \{1, 2, 3\} \\
 S_4 &= \{1, 2, 4\} \\
 S_5 &= \{1, 5, 6\} \\
 S_6 &= \{1, 5, 7\} \\
 S_7 &= \{1, 5, 7\}.
 \end{aligned}$$

```

1 public static final int THRESHOLD = 15; // <1 min with 40 cores
2 public static Set<Coverage> solve(List<Coverage> coverage) {
3     final Set<Coverage> solution = new HashSet<>();
4     List<Coverage> rest = new ArrayList<>(coverage);
5     while (!rest.isEmpty()) {
6         while (!rest.isEmpty()) { // cheaper steps first
7             final int restSize = rest.size();
8             final int size = solution.size();
9             rest = removeDuplicates(rest); // O(n)
10            rest = removeMustHaveItems(rest, solution); // O(n)
11            if (rest.size() == restSize && size == solution.size())
12                break; // nothing changed
13        }
14        final int restBefore = rest.size(); // O(n^2)
15        rest = removeSubsets(rest);
16        if (restBefore == rest.size())
17            break; // no reduction possible anymore
18    }
19    if (!rest.isEmpty() && (rest.size() <= THRESHOLD))
20        solution.addAll(fullSearch(rest));
21    if (rest.size() > THRESHOLD) // optimality not guaranteed
22        solution.addAll(greedy(rest));
23    return solution;
24 }

```

Figure 3.11: Algorithm to solve the set cover problem starting with logical reductions.

We follow the algorithm as shown in Fig. 3.11 and obtain the following results after each step:

Step 1	Step 2	Step 3
$U = \{2, 3, 4\}$	$U = \{2, 3, 4\}$	$U = \{\}$
$F = \{S_1, S_2, S_3, S_4\}$	$F = \{S_1\}$	$F = \{\}$
$S_1 = \{2, 3, 4\}$	$S_1 = \{2, 3, 4\}$	
$S_2 = \{3, 4\}$		
$S_3 = \{2, 3\}$		
$S_4 = \{2, 4\}$		
$S_p = \{S_5, S_6\}$	$S_p = \{S_5, S_6\}$	$S_p = \{S_1, S_5, S_6\}$
$\bigcup_{S_p} = \{1, 5, 6, 7\}$	$\bigcup_{S_p} = \{1, 5, 6, 7\}$	$\bigcup_{S_p} = \{1, 2, 3, 4, 5, 6, 7\}$

In the first iteration of the logical reduction, we remove the duplicate  $S_7$  from  $F$  and then identify the elements 6 and 7 as must-have items. We include the sets with elements 6 and 7, i.e.,  $S_5$  and  $S_6$ , into the partial solution  $S_p$ . Then, we also remove  $S_5$  and  $S_6$  from  $F$ . Finally, we remove all elements in sets in  $S_p$ , i.e.,  $\{1, 5, 6, 7\}$  from  $U$  and from all  $S_i$  not in  $S_p$ . Step 1 presents the reduced problem.

In the second iteration of the inner loop, we do not find any duplicates or must-have items. Therefore, the program execution continues with the detection of subsets. We identify  $S_2$ ,  $S_3$ , and  $S_4$  as subsets of  $S_1$  and remove them. Step 2 presents the reduced problem.

The next iteration of the inner loop identifies the elements 2, 3, and 4 as must-have items. We add  $S_1$  to  $S_p$ , remove  $S_1$  from  $F$  and remove all must-have items that we found from  $U$  and from all  $S_i$  not in  $S_p$  (none in this case). Step 3 presents the reduced problem. We notice that the problem is now an empty problem and therefore we found an optimal solution. Due

to limitations in space and brevity, we do not present an example where the remaining problem size requires further enumeration or the greed approach.

In practice, we were able to solve all problem instances we encountered with a (provable) optimal solution in a reasonable amount of time. Over a time frame of 3 years, we encounter problem instance with sizes  $3\,000\,000 \leq |U| \leq 5\,000\,000$ ,  $1\,000 \leq |F| \leq 3\,000$ , represented by 50 GB to 200 GB of data. Even in larger instances, the execution time for an implementation of our approach is typically below 1 minute, and always below 5 minutes (80 cores, 3 GHz, and 128 GB RAM) – for a provable optimal solution.

Effective for large problem instances in practice

But even with these promising practical results, the overall problem remains NP-hard. We can trivially construct a problem instance that we are unable to reduce logically with our reduction steps. Assume that

$$n, m \in \mathbb{N}^+ \text{ with } n > 2, m = n - 1 \quad (3.68)$$

$$U = \{0, 1, 2, \dots, n - 1\} \quad (3.69)$$

$$F = \{S_0, S_1, \dots, S_m\} \quad (3.70)$$

$$S_i = \{i \bmod n, (i + 1) \bmod n, \dots, (i + m - 1) \bmod n\}. \quad (3.71)$$

For such problem instances, it follows that:

$$\forall X, Y \in F : X \not\subseteq Y \quad (3.72)$$

$$\forall e \in U : |\{S \mid e \in S, S \in F\}| = n - 1 > 1. \quad (3.73)$$

We can conclude that Eq. (3.72) prevents any reduction by *duplicate/subset removal*, and Eq. (3.73) prevents any reduction by *must-have item removal*. Therefore, we cannot reduce such problems. With, say,  $n > 100$ , we are also unable to enumerate all solutions in a reasonable amount of time. However, as previously noted, the structure of practical problem instances is typically suitable for our approach.

The logical reductions we described are well-known in related work [161]. Our practical contributions in this case are:

1. An implementation utilizing efficient data structures and parallel algorithms with polynomial worst-case algorithmic time complexities.
2. The repeated application of a combination of the reductions.
3. An evaluation and experience report with practical problem sizes that contain millions of elements and thousands of sets.

We also considered an additional logical reduction, namely *partition detection*, where we partition the original problem in two separate smaller problems if possible. This is possible if we can find  $U_1, U_2 \subset U, F_1, F_2 \subset F$  where  $U_1 \cup U_2 = U$  and  $F_1 \cup F_2 = F$  such that  $U_1 \cap U_2 = \emptyset, F_1 \cap F_2 = \emptyset, x_1 \notin S_i \forall x_1 \in U_1 \forall S_i \in F_2$ , and  $x_2 \notin S_i \forall x_2 \in U_2 \forall S_i \in F_1$ . Informally, we search for non-overlapping partitions. We expect to find such partitions in  $\mathcal{O}(m)$ , where  $m = |F|$ , with an algorithm that transforms the sets into a graph and tests for graph connectivity. The graph consists of nodes that are the elements of  $U$  and edges that represent membership in a set  $S_i$ , i.e., if  $e_1, e_2 \in S_i$  then we add an edge between the nodes for  $e_1$  and  $e_2$ . In such a graph, we can test for reachability by utilizing a visited attribute (see related work for graph connectivity [75]). However, we did not encounter any

Partition detection

problem instance where we could test the usefulness of this additional step, as the previous reductions already solved all practical problem instances.

We use an implementation for our approach in several cases where we can transform a problem to the set cover problem, e.g., Section 4.4 or Section 5.3.

### 3.3.6 Greedy Algorithm

For problem instances of the SCP that cannot be solved by only logical reductions, we need an additional approach. A simple, but effective approach in terms of execution time and accuracy, is a greedy algorithm. It follows the core idea that, given a problem instance with several sets, the algorithm selects a set to include into the solution based on a heuristic. The most simple heuristic rates each set by the number of elements that are not yet included within a (partial) solution. Based on this heuristic, the greedy algorithm selects the set with the largest number of new elements. Fig. 3.12 shows an implementation.

Select the next available set with the highest gain

```

1 /** Solves the SCP. Fast, but may not to be optimal. */
2 public List<Coverage> greedy(final List<Coverage> cov) {
3     Coverage solution = Coverage.EMPTY;
4     List<Coverage> all = copySort(all, sortByCardinalityDesc);
5     for (int i = 0; i < all.size(); i++) {
6         final Coverage currentItem = all.get(i);
7         if (currentItem.isEmpty())
8             break; // sorted guarantees that rest is also empty
9         solution = solution.add(currentItem);
10        List<Coverage> subList = all.subList(i+1, all.size());
11        subList.replaceAll(item -> item.sub(currentItem));
12        Collections.sort(subList, sortByCardinalityDesc);
13    }
14    return convertSolution(solution);
15 }

```

Figure 3.12: Greedy algorithm for the set cover problem.

As already discussed, the greedy algorithm execution time is in  $\mathcal{O}(m)$  where  $m = |F|$  and the algorithm achieves an approximation ratio of  $H(n)$  where  $n = |U|$  and  $H(n)$  is the  $n$ -th harmonic number [48], i.e.

$$H(n) = \sum_{i=1}^n \frac{1}{i} \leq \ln(n) + 1. \quad (3.74)$$

#### 3.3.6.1 Adaptations for the WSCP

Considering the weighted version of the set cover problem, all items have an additional attribute, the weight (or “cost”). This requires several adaptations to the approaches for the SCP. However, the core concepts remain the same. The logical reductions *duplicate removal* and *must-have item identification* remain conceptually unchanged for the WSCP. The former includes the weight in a duplicate check (i.e., keep the set with the smallest weight) and the later only considers lines. Therefore, we can (with the help of polymorphism in Java) continue to use the same source code as presented in Fig. 3.6 and Fig. 3.10.

For the *subset removal*, however, the algorithm must consider the weight before removing a subset. Only in the cases where the weight of a subset is not smaller than the weight of the superset, a subset can be removed

Subset removal must be adapted.

without altering the original problem. The following example illustrates the issue that arises if the subset has a smaller weight:

$$U = \{1, 2\} \quad (3.75)$$

$$F = \{S_1, S_2, S_3\} \quad (3.76)$$

$$S_1 = \{1, 2\} \quad w_1 = 3 \quad (3.77)$$

$$S_2 = \{1\} \quad w_2 = 1 \quad (3.78)$$

$$S_3 = \{2\} \quad w_3 = 1 \quad (3.79)$$

Removing subsets  $S_2$  and  $S_3$  results in a solution  $S_l = \{S_1\}$  with a total weight of 3. However, the weight of  $S_l$  is larger compared to the (minimal) solution  $S_m = \{S_2, S_3\}$  with a total weight of 2.

Therefore, we adapt the `isNoSubset` function in Fig. 3.7 to verify the weight condition as shown in Fig. 3.13. With these adaptations, we can again combine all logical reductions as described in Section 3.3.5. Next, we modify the heuristic of the greedy algorithm to consider weights.

```

1 public boolean isNoHeavierSubSet(CoverageWithWeight set, List<
    CoverageWithWeight> all) {
2     for (final CoverageWithWeight other : all) {
3         if ((set.weight >= other.weight)&&(set.isSubSetOf(other)))
4             return false;
5     }
6     return true;
7 }

```

Figure 3.13: Identification of coverage subsets adapted for weights.

The greedy approach in Fig. 3.12 orders the list of unused items by cardinality. However, for the weighted set cover problem, we adapt the heuristic by considering the Weight of the items. An item  $i_1$  with cardinality  $|i_1| = 10$  and weight  $w_2 = 2$  may be favored in comparison to an item  $i_2$  with  $|i_2| = 20$  and  $w_2 = 20$ . Therefore, we order the list of unused items by the ratio of cardinality over weight. Fig. 3.14 shows an implementation of this adapted approach.

Adapt heuristic to use the ratio cardinality over weight

```

1 /** Solves the WSCP. Fast, but may not to be optimal. */
2 public static List<CoverageWithWeight> greedy(final List<
    CoverageWithWeight> cov) {
3     CoverageWithWeight solution = CoverageWithWeight.EMPTY;
4     List<CoverageWithWeight> all = copySort(all,
        sortByCardinalityOverWeightDesc);
5     for (int i = 0; i < all.size(); i++) {
6         final CoverageWithWeight currentItem = all.get(i);
7         if (currentItem.isEmpty())
8             break; // sorted guarantees that rest is also empty
9         solution = solution.add(currentItem);
10        List<CoverageWithWeight> sub=all.subList(i+1,all.size());
11        sub.replaceAll(item -> item.sub(currentItem));
12        sort(sub, sortByCardinalityOverWeightDesc);
13    }
14    return convertSolution(solution);
15 }

```

Figure 3.14: Greedy algorithm for the weighted set cover problem.

Finally, we can conceptually use the same approach for the full solution as in Fig. 3.11. We adapt the data type and use the algorithms provided by



Fig. 3.13 and Fig. 3.14. We omit the presentation of the adapted implementation due to a large amount of redundancy with Fig. 3.11 and the absence of any novel aspects in the adaption of the implementation.

The adaptations for the weighted version do not change the worst-case algorithmic time complexity. In practice, similar to our approach for the SCP, our adapted approach for the WSCP solves all instances we encountered in a short execution time with a provable optimal solution, i.e., the logical reductions and enumerating the (small) problem sizes that were left solved all instances we encountered. Therefore, we again did not investigate further optimizations for the greedy approach. However, we also executed the greedy approach for unreduced problems where we did not apply any logical reductions to evaluate its effectiveness. Even in such large problem instances with 2000 coverage files ( $F$ ) and  $5 \times 10^6$  lines of code ( $U$ ), the greedy approach is quite effective. The execution time is below 10s and the optimality is in 9 out of 10 evaluation cases perfect. However, we would not know that a solution provided by the greedy algorithm is perfect (or not) without the additional information.

In conclusion, due to the fast execution times of our full approach and the additional information gained about the optimality of the solution, we prefer to use our full approach instead of only the greedy approach even if the greedy approach is also rather effective. We continue to use the greedy approach as a fall-back in cases our logical reductions are unable to reduce a problem below our stated threshold.

### 3.3.7 Further Variants of the Set Cover Problem

For our work on large software projects, we encountered several other variants of the SCP. We discuss these variants of the SCP and show practical approaches and implementations to solve them. The literature provides a wide range of additional variants of the SCP. For example, Cohen and Katzir provide a discussion of the generalized maximum coverage problem that includes several other variants of the SCP [50]. Also, the original work of Karp serves as a suitable starting point to find additional related work [148].

#### 3.3.7.1 Weighted Set Cover Problem with a Budget

The WSCP with a budget (BWSCP) is a further variant of the WSCP. In contrast to the WSCP, a threshold limits the total weight of the solution. We search for the subset that covers the largest amount of items with a weight not exceeding the threshold. More formally, given a threshold  $t \in \mathbb{N}_0$ , and a universe  $U$  of  $n \in \mathbb{N}_0$  elements, and a family  $F$  of  $m \in \mathbb{N}_0$  subsets of  $U$  with associated weights, so that the union of all subsets is the universe:

$$t \in \mathbb{N}_0 \tag{3.80}$$

$$U = \{e_1, e_2, \dots, e_n\} \tag{3.81}$$

$$F = \{S_1, S_2, \dots, S_m\} \tag{3.82}$$

$$\mathbb{W} : F \rightarrow \mathbb{N}^+ \text{ is a weight function} \tag{3.83}$$

$$S_i \subseteq U \text{ and } \bigcup_{S \in F} S = U. \tag{3.84}$$

WSCP with threshold

The BWSCP asks to find a subset  $R$  of  $F$  where the sum of the weights does not exceed the threshold  $M$  and the amount of items in  $R$  is maximal:

$$\text{Find } R \subseteq \{1, 2, \dots, m\} \quad (3.85)$$

$$\text{such that } \sum_{i \in R} w(S_i) \leq t \quad (3.86)$$

$$\text{and } \left| \bigcup_{i \in R} S_i \right| \geq \left| \bigcup_{i \in X} S_i \right| \quad (3.87)$$

$$\forall X \subseteq \{1, 2, \dots, m\} \text{ where } \sum_{i \in X} w(S_i) \leq t. \quad (3.88)$$

The BWSCP solves the practical problem of selecting a set of tests that execute within a given amount of time (a *time budget*) and provide the highest amount of coverage in this time. Consider the following example:

$$U = \{1, 2, 3, 4, 5\}, \quad F = \{S_1, S_2, S_3, S_4, S_5\}, \quad t = 60 \text{ (minutes)}$$

$$S_1 = \{1, 2, 3, 4, 5\} \quad w(S_1) = 90 \quad S_4 = \{1\} \quad w(S_4) = 15$$

$$S_2 = \{1, 2, 3\} \quad w(S_2) = 40 \quad S_5 = \{3, 5\} \quad w(S_5) = 15$$

$$S_3 = \{4, 5\} \quad w(S_3) = 40$$

The test associated with  $S_1$  covers the complete universe, but the execution time of  $S_1$  exceeds the threshold, and therefore the test cannot be selected. Alternatively, the tests associated with  $S_2$  and  $S_3$  also cover the complete universe, but the sum of the execution time also exceeds the threshold. Therefore, the solution with the highest coverage is  $R = \{S_2, S_5\}$  with  $\bigcup_{i \in R} S_i = \{1, 2, 3, 5\}$  and a total weight of  $\sum_{i \in R} w_i = 55$ .

In contrast to the SCP and the WSCP, the logical reductions steps are only partially applicable. The *must-have item identification* is not sound due to the budget limitation, which can exclude a solution that contains all elements. We can apply *duplicate removal* and *subset removal* from WSCP. However, without the iterative identification of must-have items, the impact is rather limited for our practical test data. Due to randomness introduced by the system itself, subset relations rarely occur in the full data.

To solve the BWSCP, we can adapt the greedy algorithm similarly as for the WSCP. In contrast to Fig. 3.14, we skip items where the sum of the weights would exceed the given threshold as shown in Fig. 3.15.

In contrast to the greedy approach for the WSCP, the approximation factor of the greedy approach for the BWSCP is unbounded. To show this, we follow an analysis for the budgeted maximum coverage problem (BMCP)<sup>2</sup> [156]. For this purpose, we use the adapted example:

$$\begin{aligned} t &= p + 1, p \in \mathbb{N}^+ \\ U &= \{x_1, \dots, x_p, x_{p+1}\} \\ F &= \{S_1, S_2\} \\ S_1 &= \{x_1, \dots, x_p\} w(S_1) = p + 1 \\ S_2 &= \{x_{p+1}\} \quad w(S_2) = 1. \end{aligned}$$

The greedy heuristic calculates  $|S_1|/w(S_1) = p/(p + 1) = k_1 < 1$  and  $|S_2|/w(S_2) = 1 = k_2$ . Based on  $k_1 < k_2$ , the greedy heuristic selects  $S_2$  and

Logical reduction are not effective for BWSCP.

<sup>2</sup> The BMCP is in fact identical to the BWSCP except that each element in  $U$  has a specific weight (or benefit) where for the SCP and BWSCP, each element in  $U$  has an uniform weight of 1 [156].

```

1 /** Solves the BWSCP. Fast, but may not to be optimal. */
2 public static List<CoverageWithWeight> greedy(final List<
3     CoverageWithWeight> cov, final int maxWeight) {
4     CoverageWithWeight solution = CoverageWithWeight.EMPTY;
5     List<CoverageWithWeight> all = copyAndSort(cov,
6         compareByCardinalityOverWeightDesc);
7     for (int i = 0; i < all.size(); i++) {
8         final CoverageWithWeight current = all.get(i);
9         if (current.isEmpty())
10            break; // sorted guarantees that rest is also empty
11         if (solution.getWeight()+current.getWeight() > maxWeight)
12            continue;
13         solution = solution.add(current);
14         List<CoverageWithWeight> sub=all.subList(i+1,all.size());
15         sub.replaceAll(item -> item.sub(current));
16         sort(sub, sortByCardinalityOverWeightDesc);
17     }
18     return convertSolution(solution);
19 }

```

Figure 3.15: Greedy algorithm for the weighted set cover problem with a budget.

reports the solution  $R_2 = \{S_2\}$  with  $|R_2| = 1$  where the optimal solution is  $R_1 = \{S_1\}$  with  $|R_1| = p$ . The approximation is  $p$  and unbounded.

Khuller et al. report two modified greedy algorithms, *KA1* and *KA2*, with approximation factors of  $(1 - 1/e)/2$  for *KA1* and  $(1 - 1/e)$  for *KA2* [156]. Both *KA1* and *KA2* require additional calculations to improve the approximation factor. *KA1* calculate the best single-element solution over all elements in  $F$  and returns this solution if better than the greedy approach. *KA1* calculates  $|F|$  greedy results where each item in  $F$  is pre-selected for a calculation and then reports the best result. In practice, we did not encounter any problem instance where *KA1* or *KA2* provided better results than our described approach. Even more, the execution times of *KA2* exceeded 10 min in several cases, making it uninteresting for practical cases.

### 3.3.7.2 Set Cover Problems With Multiple Buckets

Large projects utilizes multiple servers for test executions, say a set  $SV = \{SV_1, \dots, SV_n\}$  of  $n$  servers. This leads to the question of how to distribute the test executions over the servers in  $SV$ . Our work does not focus on this distributed aspect. However, to show that our algorithms can be adapted for such cases, we present the ideas for two approaches, a first-come-first-serve approach and a multiple bucket variant of the greedy approach.

SCP with multiple buckets

For the first-come-first-serve approach (FCFS), we apply the greedy approach for the BWSCP to obtain a solution  $R$  with an ordered sequence of tests. Then, for each server  $SV_i$  in  $SV$  that is not executing a test, we obtain the first test  $t_{top}$  from  $R$ , remove it from  $R$  and execute  $t_{top}$  on  $SV_i$ . We repeat this approach until all tests in  $R$  are executed.

For the multiple bucket variant of the greedy approach, we modify the greedy algorithm for the BWSCP to include buckets as shown in Fig. 3.16.

The FCFS approach even supports a variable amount of servers. However, the distribution can be unbalanced where one server executes a test with a large execution time while all other servers are finished and idle. In practice, parallel test runs mitigate this issue. The greedy with multiple buckets approach aims to balance the test executions and therefore can reduce the

```

1 /** Solves the BWSCP with buckets. Fast, but may not to be
   optimal. */
2 public static List<CoverageWithWeight> greedy(List<
   CoverageWithWeight> cov, int maxWeight, int buckets) {
3     final List<T> solutions = new ArrayList<>(Collections.
   nCopies(buckets, CoverageWithWeight.EMPTY));
4     final List<CoverageWithWeight> all = copySort(cov,
   sortByCardinalityOverWeightDesc);
5     for (int i = 0; i < all.size(); i++) {
6         final CoverageWithWeight current = all.get(i);
7         if (current.isEmpty())
8             break; // sorted guarantees that rest is also empty
9         for (int n = 0; n < solutions.size(); n++) {
10            CoverageWithWeight solution = solutions.get(n);
11            if (solution.getWeight()+current.getWeight() > maxWeight)
12                continue;
13            solutions.set(n, solution.add(current));
14            List<CoverageWithWeight> sub=all.subList(i+1,all.size());
15            sub.replaceAll(item -> item.sub(current));
16            sort(sub, sortByCardinalityOverWeightDesc);
17            break;
18        }
19    }
20    return convertSolutions(solutions);
21 }

```

Figure 3.16: Greedy algorithm for the weighted set cover problem with a budget and buckets.

total idle times of all servers for a single test run. This typically also results in a shorter test execution time. However, the planning for parallel test runs gets rather complex with such an approach and therefore the FCFS approach is preferred in practice for SAP HANA.

We visualize the differences by an example. Given 10 tests with test execution times  $\{1, 4, 9, 5, 1, 6, 6, 9, 10, 4\}$  (in this order), we apply both approaches to distribute the tests over 4 servers. For the greedy algorithm, we assume a  $\text{maxSize} = 15$ . Fig. 3.17 shows the results. We measure the total execution time  $T_m$ , which is the maximum of all bucket sums and represents the time until the test run is finished. The FCFS approach then results in a distribution with  $T_m = 1 + 1 + 6 + 10 = 18$ . However, the greedy approach finds  $T_m = 1 + 4 + 9 + 1 = 15$ .

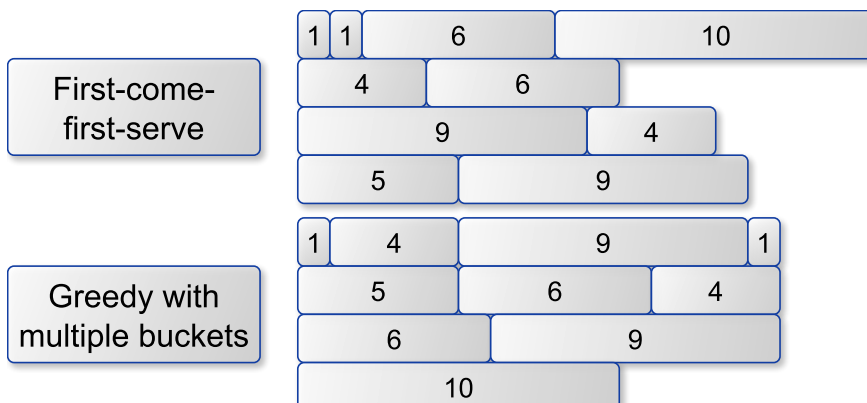


Figure 3.17: A comparison between two approaches for distributing tests with different execution times over multiple servers.

The theoretical problem behind finding a distribution of  $n$  tests, each with an execution time  $t_i$ , over  $m$  servers that satisfies predefined requirements is a variant of the *bin packing problem* [162] (BPP) and the *multiprocessor*

Bin packing problem

*scheduling problem* [97]. The BPP consists of  $n \in \mathbb{N}^+$  items with associated sizes  $a_1, \dots, a_n$  and a set  $B$  of  $k \in \mathbb{N}^+$  bins  $\{B_1, \dots, B_k\}$  with the same size  $b \in \mathbb{N}^+$ . Then, for a solution, all items must be distributed over all bins without violating anysize constraints. More formally:

$$\exists f : \{1, \dots, n\} \rightarrow \{1, \dots, k\} : \forall j \in \{1, \dots, k\} : \sum_{i:f(i)=j} (a_i \leq b) \quad ? \quad (3.89)$$

In our case, the bins represent servers and the items with sizes are tests with execution times. Also, the bin size is variable, but the amount of bins is bounded. We can map our problem to the BPP by introducing a threshold for execution times. Then, our greedy approach is identical to the *first-fit* algorithm with an approximation factor of 2 of related work [162].

### 3.3.8 Coverage Compaction

Epitropakis et al. propose an algorithm to compact coverage [85]. The algorithm is lossless, i.e., it allows us to reconstruct the original data. Hence, we can use compacted coverage used instead of the original data. The compaction reduces space requirements, and can potentially reduce the execution time of any algorithm working on coverage data.

Lossless compaction

The main idea for the algorithm is to identify two parts of the code that show exactly the same behavior in terms of coverage for all test executions. We can then remove one of these parts because its behavior is determined by the other part. More formally, given the set of all tests  $S_T$ , the set of all source code lines  $S_L$  and the coverage function  $cov_i(l)$  that reports the coverage of a source code line  $l \in S_L$  for the execution of  $i \in S_T$ , then:

$$\text{Find pairs } (k, l) \in S_L \times S_L : cov_i(k) = cov_i(l) \forall i \in S_T. \quad (3.90)$$

If such a pair exists, we say that  $k$  relates to  $l$ , or  $k \sim l$ . The set of all such pairs represent a binary relation  $R_C$ :

$$R_C = \{(k, l) \mid cov_i(k) = cov_i(l) \forall i \in S_T \text{ with } k, l \in S_L\}. \quad (3.91)$$

We can see that  $R_C$  is reflexive, symmetric and transitive. Hence,  $R_C$  is an equivalence relation [230]. We can partition  $R_C$  into a set of equivalence classes  $EC_{RC} = \{[a_1], \dots, [a_n]\}$ , where  $[a_i] = \{x \mid x \sim a_i, x \in S_L\}$ . Then,  $[k] = [l] \iff k \sim l$  and we select  $[min(k, l)]$  as the canonical representative for the equivalence class. Given these definitions, we can understand the compaction algorithm as selecting representatives of the equivalence classes produced by the relation  $R_C$  and thereby reducing  $S_L$  to  $EC_{RC}$  with  $|EC_{RC}| \leq |S_L|$ . In practice,  $|EC_{RC}| \ll |S_L|$ , i.e., the set of representatives is smaller by a large factor than the set of all lines.

Equivalence relation

We highlight two improvements for the original ‘‘Algorithm 1’’ ( $ALG_{cce}$ ) by Epitropakis et al. [85]. First, the description of  $ALG_{cce}$  has a small flaw in line 7. The delete operation invalidates the index. This is easily fixed by a temporary variable. Most likely, the authors implemented this fix for their evaluation. Second, the worst-case time complexity for  $ALG_{cce}$  is quadratic:  $\mathcal{O}(n^2)$  where  $n = |S_L|$ , i.e., the number of lines. For practical cases with several million lines, the quadratic complexity results in infeasible execution times. Therefore, we use hash-based filtering which leads to an amortized [241] linear time complexity  $\mathcal{O}(n)$  as shown in Fig. 3.18.

```

1 public List<Coverage> compactCoverage(List<Coverage> all) {
2     Coverage sum = parallelSum(all);
3     List<SourceLinePair> slps = sum.getSourceLinePairs();
4     Set<BitSet> unique = slps.parallelStream().map(slp ->
5         getEcForCoveragesAndSlp(all, slp)).collect(toSet());
6     // #unique: unique.size(), #redundant: slps.size() - unique
7     return convert(all, unique) //converts the result to list
8 }
9 public BitSet getEc(List<Coverage> all, SourceLinePair slp) {
10     final BitSet result = new BitSet(all.size());
11     for (int i = 0; i < all.size(); i++) {
12         if (all.get(i).isSourceLineHit(slp))
13             result.set(i);
14     }
15     return result;
16 }

```

Fig. 3.20 presents an example. The first three lines in Fig. 3.20a may represent code as shown by Fig. 3.19. Fig. 3.20 also shows multiple variants. The variant selection depends on the requirements of the next step. For example, our approaches to the set cover problem presented in Section 3.3.4 do not require the identification of specific lines. Therefore, we can utilize Variant B as presented in Fig. 3.20c where we re-label the lines with a new counter and further reduce the coverage data size compared to variant A.

Test	Lines					
	1	2	3	4	5	6
A	x	x	x	x		x
B	x	x	x		x	
C				x	x	x

(a) Original coverage data.

Lines	A	B	C	Lines	A	B	C
1, 2, 3 → 123	3	3		1, 2, 3 → 123	x	x	
4, 6 → 46	2		2	4, 6 → 46	x		x
5 → 5		1	1	5 → 5		x	x

(b) Compacted coverage variant A.

(c) Compacted coverage variant B.

### 3.4 Summary

We introduced the concept of coverage and defined all related terms. The exact definitions of coverage are important because the term is used in literature with different meanings. In our work, we focus on line coverage data. However, several of our approaches also support other variants.

We also presented approaches for several algorithmic problems when working with coverage data. We will use them as building blocks for advanced approaches. One of the design goals is scalability for large projects. Based on practical experience and evaluations, this design goal is achieved. We can run our analysis, experiments, and algorithms on coverage data in short times. This allows a practical adoption and rapid exploration of new ideas.

Figure 3.18: Hash-based compact coverage algorithm.

```

1 int f(int a, int b) {
2     int c = a + b; // line 1
3     int d = a - b; // line 2
4     return c*d;    // line 3
5 }
6 // ... line 4,5,6

```

Figure 3.19: Example where executions result in redundant line coverage.

Figure 3.20: Example for coverage compaction. Table (a) indicates for 3 tests which of 6 lines are executed (“x”). Table (b) shows the compacted version where 3 lines are removed, but all information is preserved. Each number indicates how often the corresponding group of lines (indicated by the label) is executed. Table (c) shows a further simplified variant, where only a single bit for the execution state is required instead of the execution count. The label for the lines may be used to extract all required information to reconstruct the original state. Finally, we may also replace the labels with a new counter if reconstruction is not required.

# 4 | On the Relationship Between Coverage and Faults

Code coverage data for a test execution provides information about the executed parts of a software  $P$ . This means that we can derive from coverage data which parts of the source code were executed by a test and which parts were not. We also discussed in Chapter 2 that one of the purposes of testing is to show the presence of defects in source code<sup>1</sup>. As both concepts provide information about source code, we can consequently ask whether coverage data provides any additional information on faults.

To understand this question, we first discuss the essence and limitations of information gained by coverage data. We then discuss related work and possible implications. Finally, we contribute two empirical studies with new insights on the relationship between coverage and faults. More specifically, our contributions are:

- A study that investigates the impact of coverage on the distribution of defects for a large industrial system (Section 4.2).
- A study that investigates Granger-causality between coverage and defects for a large industrial system and an open-source project (Section 4.3).
- An approach to combine coverage-based input partition and combinatorial testing to increase coverage (Section 4.4).

## 4.1 Discussion

Code coverage measures to which degree a software under test (SUT) is executed by software tests. For instance, a set of software tests may execute 80% of a SUT and therefore the SUT has 80% test coverage. What do we learn from such a number? It would be a fallacy to conclude that 80% of this SUT is free of faults. In fact, tests are unable to prove the absence of faults, they only prove their presence. We have no information about the total amount of faults in a software project, therefore we cannot conclude whether we found all failures. Furthermore, it may not be possible to derive the total number of faults. Typically, faults exist because we are not aware of them. Otherwise, we would just correct them. To derive the number of all faults, we would require to have information about the information we do not know. Without an external source of information, this seems infeasible.

Practically speaking, it is a typical experience of every developer that faults reveal at surprising and unexpected places. For example, binary search is a fundamental algorithm [53] for which it seems to be a rather simple task to create, understand, and verify an implementation. Yet, it took 20 years to detect a fault in the implementation for the Java standard library [227].

<sup>1</sup> It is worth mentioning that testing does not show the absence of defects – a quote attributed to Edsger W. Dijkstra.

Tests are unable to prove the absence of faults, they only prove their presence

Fig. 4.1 shows the original Java code containing the fault caused by an integer overflow [227]. As this example already shows, most likely no person with experience in software engineering would claim that software contains no faults based on a set of (succeeding) tests.

Therefore, we cannot make a statement about the number of faults for the 80% of covered source code. However, for 20% of the SUT, we can clearly state that we know nothing. These 20% were not executed at all and, therefore, they are not tested. This statement seems to be trivial and obvious. However, it provides value for our testing process. We can conclude that this 20% of the source code can contain any fault — the code may not work at all. Therefore, we potentially have a clear sequence of steps to improve the quality. Investigating the unknown 20% can already improve the quality of the software by only adding information about these 20% of the software.

After investigating the tested and untested parts of the software independently, we can also try to compare both parts by various metrics. By comparing the tested part of the software to the untested part of the software, we can create the hypothesis that the tested part of the software contains a lower amount of failures compared to the amount of failures in the untested part of software. At first, it seems plausible that this hypothesis is true. Software testing is an activity that typically results in quality improvements, therefore we would expect that the tested part of a software has a higher quality compared to the untested part of a software. However, this argument can be misleading for the following reasons:

- The untested part of the software could already have a high quality ensured by other measures than test executions. The testing process could use a risk-based approach [89] and therefore targets only the parts of the software with a high risk.
- The untested part could contain functionality that is not relevant for the product. There could be failures, but they would not appear to users. Therefore, the quality of the product is not affected by the untested part.
- The number of known failures might not be distributed evenly. Typically, we might find more failures in these parts of the software that are more frequently used. This follows by a statistical argument. The more users that use a specific functionality, the higher the chance they encounter a defect in some usage scenarios. Therefore, even if the untested part of the software is infrequently used by users, the total amount of defects encountered can be lower compared to the more frequently used (and tested) parts of the software.

Altogether, we conclude that the existence of causality between tested code and code quality remains unclear. We only know that the properties encoded in the tests and checked by the tests hold if the tests succeed. However, in practice, code coverage is used to guide testing activities and as a metric to measure the quality of a test suite (or the program itself). Therefore, we can ask whether there is a correlation between coverage or the degree of coverage and quality<sup>2</sup>. There is a wide range of related work investigating these questions. In addition to the existing work, we also conducted several studies on SAP HANA.

```

1 int bins(int [] a, int key){
2   int low = 0;
3   int high = a.length - 1;
4   while (low <= high) {
5     int mid = (low+high)/2;
6     int midVal = a[mid];
7     if (midVal < key)
8       low = mid + 1
9     else if (midVal > key)
10      high = mid - 1;
11     else //key found
12       return mid;
13   }
14   return -(low+1); //no key
15 }

```

Figure 4.1: Faulty binary search implementation. Original version of Java standard library before 2006.

Coverage can guide testing activities.

Has tested code fewer faults compared to untested code?

If no causation, is there a correlation?

<sup>2</sup> Reminder: Correlation does not imply causation.



#### 4.1.1 *Prior Work on Benefits of Code Coverage*

There are several studies that analyze whether the existence of code coverage itself and coverage goals are beneficial for measuring and improving software quality. The general hypothesis that a high test coverage results in a low amount of defects might sound trivial. Test coverage represents the execution of tests. Therefore, if a large amount of the source code is tested by tests, then it should have a low amount of defects. In fact, that is one of the reasons why we test – to detect and remove defects. While it seems to be obvious, we argue that it is in fact not. We discuss two cases: a relative comparison, and an absolute statement.

In the relative case, we argue that it is unclear whether an increase of  $X\%$  in test coverage results to a decrease of  $Y\%$  in defects. It may happen that  $X$  occurs at trivial places (say, trivial getter/setter methods for objects) and therefore it has no implications on the number of defects. Then we could ask how large has  $X$  to be that  $Y$  will be influenced? Presumably, there are diminishing returns – if not, all defects would be found at  $X = 100$ . Maybe there is even a global maximum or plateau. In a more extreme case, we could add tests that execute the software but do not verify the results (i.e., they do not contain a test oracle). This would increase coverage, but clearly provide little benefit for finding defects. We conclude that it requires further empirical analysis to draw any conclusion in this case.

An increase in coverage must not reduce the number of defects.

In the absolute case, we argue that it is unclear how to interpret a ratio of  $X\%$  in test coverage regarding the number of defects. In fact, how should we draw any conclusion at all? The total number of defects is not known. Maybe the number of defects found is known (although we see later that even this is typically not known), but even then how should we correlate both numbers, coverage and defects? We could try to start with the most simplified case where  $X = 0$ . As we discussed in Chapter 3, the only valid conclusion we can make in this case is that the number of defects is unknown. It might be 0 or any other number. Conclusively, it requires further empirical analysis to draw any conclusion in this case.

A coverage ratio does not provide information about the number of defects.

Given these arguments, we conclude that the question of whether a high test coverage results in a low amount of defects is in fact not trivial to answer. A sound answer requires empirical research and statistical arguments.

#### 4.1.2 *Results Supporting Benefits of Code Coverage*

These publications often study the correlation between code coverage ratio and the number of bugs in different test subjects. For example, Mockus et al. investigate the correlation between coverage ratio and the probability of defects affecting a component on two industrial software projects and the effort required to increase the coverage ratio [194]. They find that an increase in coverage leads to a proportional decrease in defects.

Ahmed et al. analyze 49 projects to understand the correlation between coverage and bugs [3] (see also Section 4.2.2.1). They analyze different variants of coverage and find a weak but significant correlation between statement coverage and the number of bug-fixes as a proxy for bugs.

Kochhar et al. investigate two large software systems to investigate and compute correlations between coverage, test suite size, and test suite effec-

tiveness. They find a moderate to strong correlation between coverage and test suite effectiveness [158]. A metastudy conducted by Inozemtseva et al. shows that 8 out of 12 previous studies confirm a positive correlation between coverage and test suite effectiveness [130].

#### 4.1.3 *Results Contradicting the Benefits of Code Coverage*

While other work also use correlation analysis, other authors consider additional variables in extension to only coverage and bugs. For example, Inozemtseva et al. study correlation between coverage and test suite effectiveness in five open-source systems, but they also control the test suite size for their experiments [130]. They find that it is generally not safe to assume test suite effectiveness is strongly correlated with coverage. In fact, their results indicate that test suite size is correlated with test suite effectiveness and code coverage does not show a strong correlation with test suite effectiveness when the suite size is controlled. In other words, larger test suites with more coverage can find more defects. Namin et al. find similar results for smaller study subjects and investigate multiple models for the correlation to explain such findings [199]. Groce et al. provide an extensive overview of existing work that highlights several limitations [110].

Mark Seemann claims that coverage is a “useless target measure”<sup>3</sup>. Martin Fowler states that “Test coverage is of little use as a numeric statement of how good your tests are.”<sup>4</sup>. Both practitioners argue in favor of using coverage as a helpful metric to identify untested parts of a software, but the absolute numbers have no relevance or meaning. This also implies that we would not be able to draw any conclusions regarding the number of defects.

<sup>3</sup> <http://blog.ploeh.dk/2015/11/16/code-coverage-is-a-useless-target-measure/>

<sup>4</sup> <https://martinfowler.com/bliki/TestCoverage.html>

#### 4.1.4 *Mixed Results Regarding the Benefits of Code Coverage*

Some work claim that the answer depends on the exact question, definitions of the terms, and potentially on more variables [110, 199, 232, 270]. For example, Groce et al. argue in their survey that there is uncertainty as to what exactly measuring code coverage should achieve, as well as how we would know if it can, in fact, achieve it. They develop a strong and weak coverage hypothesis to provide a meaningful context for further discussion and experiment design [110].

#### 4.1.5 *Interpretation of Results Regarding the Benefits of Coverage*

The results of related work are partially contradicting each other. While some studies show the coverage has a benefit on code quality in terms of number of defects, other studies doubt this benefit. Several studies show that the correlation seems to be spurious, i.e., there are confounding variables. In addition, some studies use only surrogates for defects, such as mutants. Therefore, we can only conclude that there is no accepted result.

This unclear result, to our belief, reflects the complexity of practical software engineering. As discussed in Chapter 2 and Chapter 3, there are several complex factors that might be hard to measure precisely:

More studies required to investigate complex factors.

1. The definition of a defect depends on the opinion of persons that classify an issue. In addition, studies typically do not investigate the impact of

defects, i.e., whether they, for instance, crash an application, alter data or only misrepresent a visual element. It remains unclear how defects could be classified and analyzed with rigor. Even if such a classification exists, the task to re-classify existing study subjects seems to be challenging. In addition, there might be several defects that exist or get fixed which are never monitored within a bug tracker or a version control system because developers fix the issue during development before integrating the source code into the main code line.

2. There are several coverage variants and it remains unclear which to investigate and how to interpret changes. Coverage might change due to several reasons such as new code, new tests, or flakiness.
3. Not all code, and therefore, not all coverage is equal. Some parts of the code might be important for a software product and are frequently used, other parts are rarely or never used. It might not be a fair comparison to investigate the complete code base assuming that all code lines are equally treated. For example, more frequently executed code has a higher chance to reveal defects compared to code that is never executed. It remains unclear how this effects a relationship between coverage and defects.
4. The programming language might influence the results. Different languages support different tooling and philosophies for software creation. While it might be simple to write unit tests and measure code coverage in Java, the same task can require considerably more effort in C++ .
5. There might be a large number of possible confounding variables, i.e., a third variable that explains the changes in coverage and defects. An example is the software development method<sup>5</sup>. In addition, the characteristics of developers affect the correlation. More experienced developers are most likely better at preventing and detecting defects. Also, they might be more motivated to write tests and to aim for test coverage.
6. The project size might influence the results. While a large project might be favorable because the results can be statistically more significant, the effort to find and analyze large projects reduces the number of large studies. Even more, large projects with over 1 000 000 lines of code and over 10 years existence might not be freely available for research.

<sup>5</sup> Although it even remains unclear which effects different software development methods have on the amount of defects.

Given these arguments, we expect that authors of a study will encounter a well-known problem of empirical studies in software engineering [91, 108, 196, 216]. Either they strictly control all variables and achieve a high construct (and internal) validity but low practicability and generalizability (i.e., low external validity), or they achieve the opposite of each. Both variants may not allow drawing practical conclusion and it remains unclear how to tackle such complex research questions.

Therefore, we follow a general approach also used in other fields. We provide additional results for the same question. This allows us to collect several such results and conduct a meta-analysis to acquire new insights and possibly draw conclusions with high construct validity but also with high external validity. To this aim, we investigate SAP HANA as a large industrial system and, in a second study, we also investigate an open-source system. In two studies, we analyze the relationship between coverage and defects for these systems.

## 4.2 The Impact of Coverage on Bug Density

Measuring the quality of test suites is one of the major challenges of software testing. To approximate test suite quality, code coverage is frequently used because it identifies tested and untested parts of the code.

Multiple previous studies have investigated the relationship between coverage ratio and test suite quality, without a clear consent in the results. We study whether covered code contains a smaller number of future bugs than uncovered code (assuming appropriate scaling). If this correlation holds and bug density is lower in covered code, coverage can be regarded as a meaningful metric to estimate the state of testing.

To this end, we analyze 16 000 internal bug reports and bug-fixes of SAP HANA, a large industrial software project. We found that the above-mentioned relationship indeed holds, and is statistically significant. Contrary to most previous work our study uses real bugs and real bug-fixes. Furthermore, our data is derived from a complex and large industrial project.

Contains covered code fewer bugs compared to uncovered code?

Real bug data instead of mutants.

### 4.2.1 Introduction

Software testing is a crucial and widely deployed tool for ensuring software quality. One of the practical challenges for software testing is measuring the quality and effectiveness of test suites. Measures for the adequacy of testing are typically used to identify whether a software artifact is not tested sufficiently well, and where improvement is needed. They also play an important role to indicate that a sufficient amount of testing has been done, and resource costs for testing eventually surpass the expected savings from reductions in the amount and impact of defects.

One of the most widely-used measures for the adequacy of testing is statement coverage. Its direct use is to identify uncovered parts of code that potentially contain further bugs, not caught by existing test code. The code coverage (ratio), i.e., ratio of covered lines to all lines, is frequently interpreted as a metric of test quality, with numerous organizations using it to set testing requirements. However, some studies question the very existence of a relationship between coverage and test suite effectiveness [130].

If a positive correlation between coverage ratio and test suite effectiveness exists, we expect to find a smaller number of future bug-fixes in the covered parts of code, see Fig. 4.2. Ahmed et al. [3] design a schema to verify the benefits of coverage as a test quality measure. Essentially, they identify the number of bugs found in covered and not covered code. If the coverage ratio is meaningless, then we expect that future bugs are distributed uniformly in covered and uncovered parts of the source code. However, if coverage ratio is meaningful, then the percentage of all future bugs found in the covered parts of the code should be *smaller* than the coverage ratio. We illustrate the *binary testedness* approach of Ahmed et al in Fig. 4.3.

Does coverage change the distribution of bugs?

While Ahmed et al. [3] use mutations as surrogates for bugs, we use records of real bugs and their bug-fixes. Another essential difference is that we collect our data from a very large industrial application with high quality requirements. Our work improves the data collection process and data evaluation by introducing multiple collection points instead of using

a single snapshot like the previous study of Ahmed et al. [3]. This reduces the risk of losing track of code changes over time and increases the size of the data set for more robust evaluation.

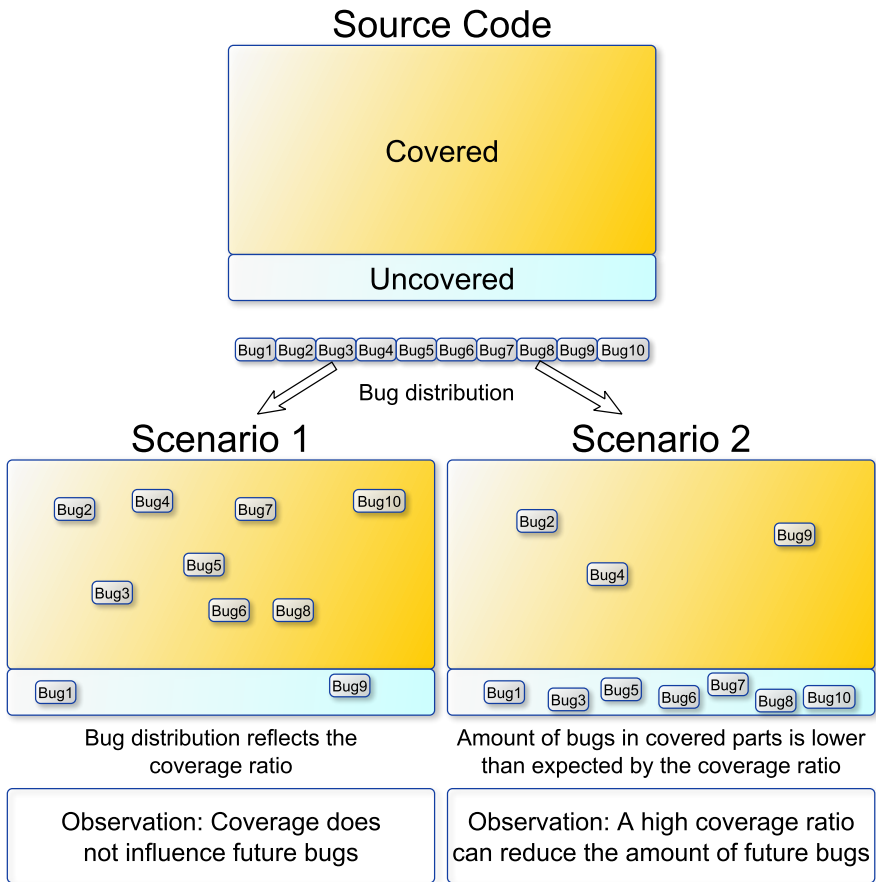


Figure 4.2: Exemplary coverage and bug distributions. Scenario 1 describes a situation where coverage ratio is meaningless for future code quality measured by the amount of bugs. Scenario 2 shows a situation where coverage is meaningful.

Following the binary testedness approach, we investigate the following research question for our study subject, SAP HANA:

RQ1 Does coverage have an impact on the distribution of detected defects?

For this purpose, we analyze how many bugs occur in covered and uncovered parts of the source code of SAP HANA. Our findings show that indeed there are fewer bugs in the covered parts of the source code than expected from a uniform distribution. This effect is visible in 70 out of 72 time segments of our data, and is statistically significant for the mean. This indicates for our test subject, that increasing coverage ratio and enforcing high coverage goals can reduce the number of defects.

Covered code has fewer bugs compared to uncovered code.

#### 4.2.2 Approach

We describe the binary testedness approach from Ahmed et al. [3], describe the SAP HANA bug tracking process environment, and outline the methodology for data collection and processing of our analysis.

4.2.2.1 Binary Testedness Approach

The binary testedness approach proposed by Ahmed et al. [3] separates the source code in two (binary) groups: Covered and uncovered parts of the source code. Based on this separation at a given time  $T$ , all future bugs and the corresponding bug-fixes after  $T$  are checked if they occur in the covered or the uncovered group, see Fig. 4.3. A lower amount of bugs in covered parts (found covered bugs) compared to coverage ratio times all bugs (expected covered bugs) would indicate that coverage is *meaningful*.

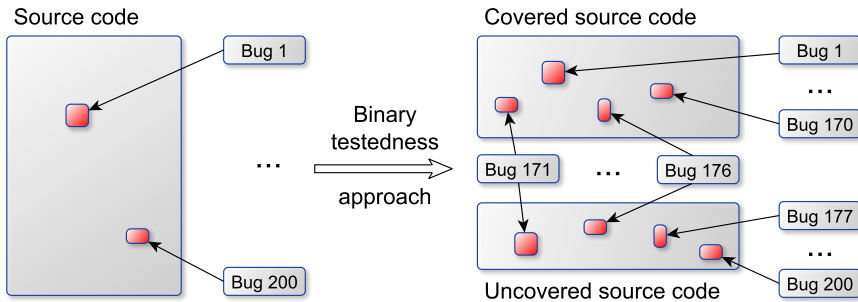
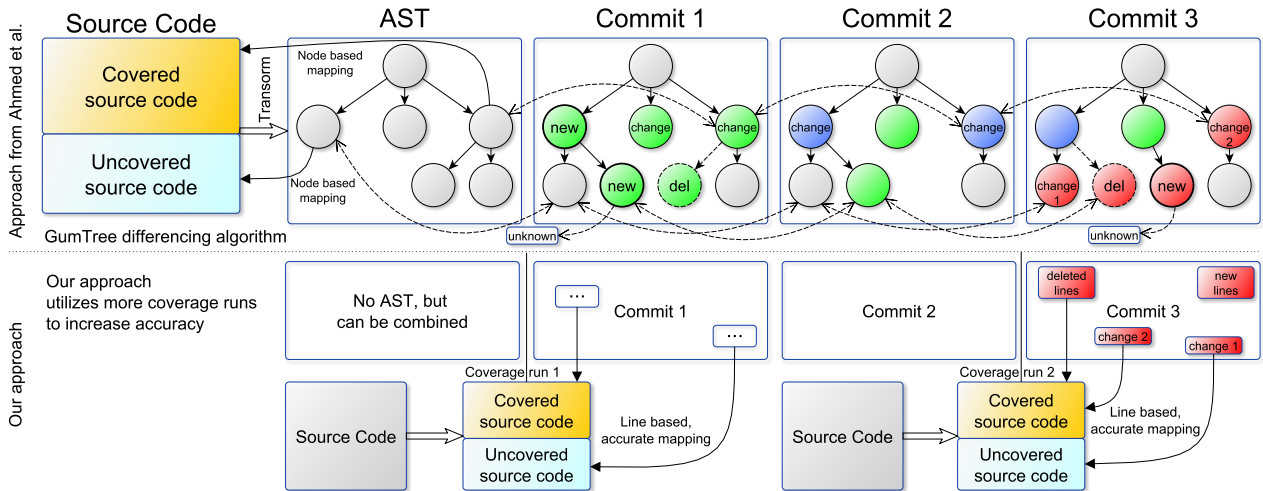


Figure 4.3: The binary testedness approach [3] separates the source code in two (binary) groups and counts the number of bugs in the covered and uncovered group. In this example, 170 bugs occurred in the covered part of the source code, 24 in the uncovered part and 6 are undecided.

Ahmed et al. use the *GumTree Differencing Algorithm* to identify the position of the original source code elements at time  $T$  for a bug-fixing commit after  $T$ , even if the source code and position of the elements changed between  $T$  and the bug-fixing commit. For this purpose, GumTree utilizes an abstract syntax tree [87], see Fig. 4.4 (upper part).

GumTree Differencing Algorithm



Instead of using coverage data for a single point  $T$  in history, our approach utilizes the data of multiple coverage runs distributed from  $T$  until the last observed bug-fixing commit. This partitions the measurement period into multiple, non-overlapping *time segments*. Fig. 4.4 (lower part) visualizes the concept and shows an example of the accuracy improvements in terms of a more robust mappings between changed source code and coverage data. Simplified, shorter time segments provide more information.

Figure 4.4: The GumTree approach [3] tracks the nodes of an abstract syntax tree (AST). This can reduce the accuracy for consecutive changes. For commit 3, change 1 is uncovered, deleted and inserted nodes are unknown and change 2 is covered, but it is, in fact, unclear due to intermediate changes. Our approach accurately maps change 1, 2 and the deletion from commit 3.

Increasing the number of coverage runs shortens the time segments, and therefore reduces the likelihood of intermediate conflicting commits between a coverage run and a bug-fix. For our study, we found that 72 time segments (roughly two coverage runs per week) ensured a mismatch rate below 1% for all file modifications. A mismatch occurs if the original source code state of the bug-fixing commit is not equal to the state of the coverage run. The accuracy can be further improved by shortening the time segments between test runs at the cost of a higher resource investment.

#### 4.2.2.2 Bug Tracking Process of SAP HANA

Section 2.2.2 describes the quality assurance process for SAP HANA. We focus here on the bug tracking process of SAP HANA.

SAP uses a bug tracking tool to maintain defect history. For the purpose of this study, we classify defects by their detection time into *early detected* and *late detected* defects. Defects found before a change is merged into the source code repository are classified as *early*. Defects found later are classified as *late*. Early defects are detected, e.g., during development, by peer reviews or by pre-commit test runs. Late defects are detected, e.g., by extended continuous automatic regression testing, by manual testing, by internal usage of SAP HANA (“self-hosting”), by automatic tools, e.g., fuzzy testing, or even by customer reports. This study focuses on late defects, which tend to have a higher cost impact [28] and better documentation compared to defects during early local development. In the following, we use the term bug synonymous with late detected defects.

Bug detected late: after code is merged into main code line.

This classification of the terms early and late bugs is a proprietary definition, which is different from common terminology, e.g., ISTQB test levels [139]. The ISTQB classification of test levels is linked to the responsibilities in a project. The distinction by responsibilities is not always possible in our case project. For instance, one test suite can contain component tests, integration tests, system tests, and regression tests. We classify bugs on the time of identification and distinguish between bugs that were detected by the current set of programmatically executed tests and bugs that were not detected by these set of tests. This allows us to apply the binary testedness approach from Ahmed et al. as described in Section 4.2.2.1.

#### 4.2.2.3 Data Collection and Processing

Our experiment setup requires coverage data, bug data, and a link between bugs and coverage. As described in Section 4.2.2.2, line-based coverage data is collected regularly. Each coverage-execution of a test suite creates a distinct coverage data file, which is aggregated with all other distinct coverage data files to a combined coverage data file.

For the bug data, we collect bugs as described in Section 4.2.2.2 and we assume that each entry in the bug tracking tool indicates a bug. Each bug entry either contains a link to one or multiple bug-fixing source code changes (“bug-fixing commit”), or the bug-fixing commit message contains the id of the bug. SAP engineers have collected, maintained, and used this information for many years within the SAP HANA project, and so we can assume that the results are reliable.

Accurate information about bugs and bug-fixing changes

This allows us to accurately identify related code changes for each bug. Conclusively, we expect our classification to be more accurate than approaches that identify such changes based on change information alone [231].

4.2.2.4 *Classifying Bug-Fixes by Coverage*

For each bug, we classify the corresponding bug-fixing as covered, uncovered or undecided. For this purpose, we use the decision graph shown in Fig. 4.5, which we explain in the following paragraphs.

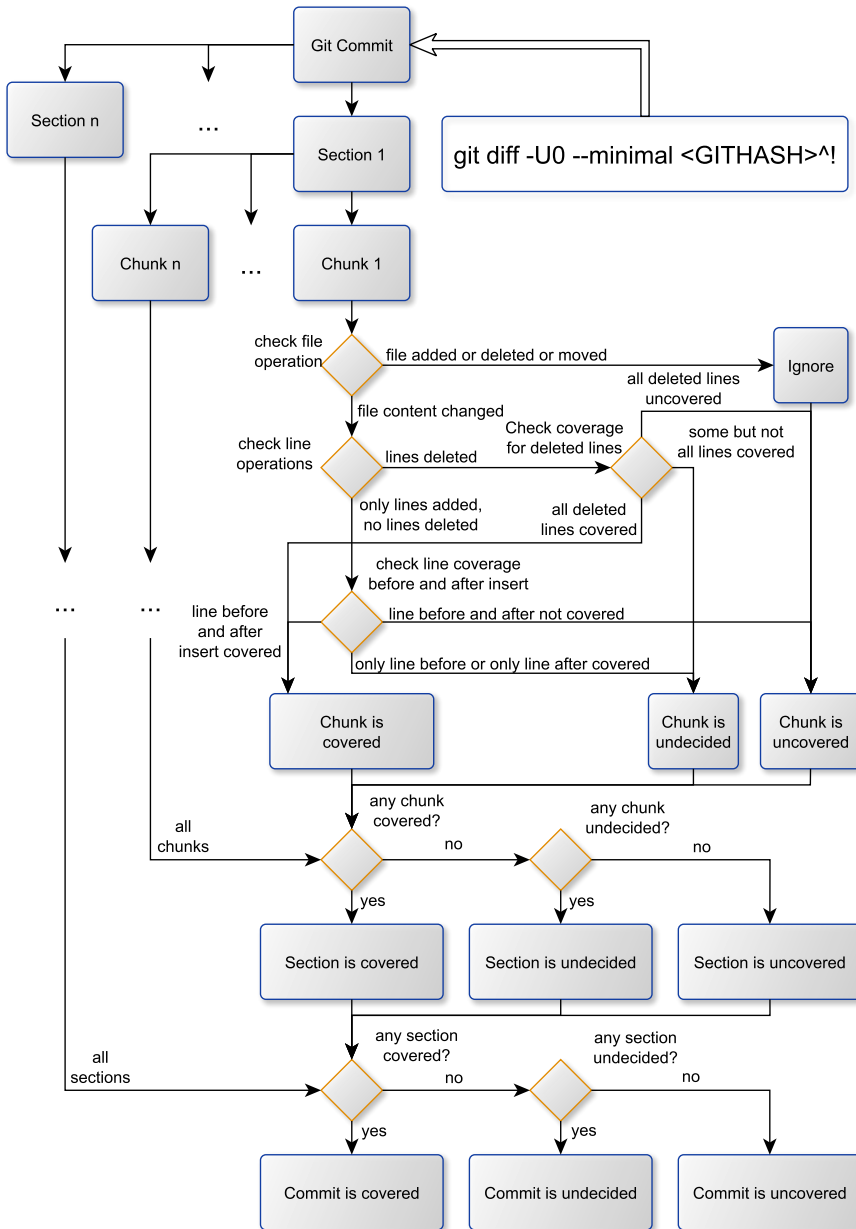


Figure 4.5: The process of classifying a bug-fixing Git commit as covered, undecided, or uncovered.

The source code of SAP HANA is maintained in a *Git* version control system [14]. Each *Git* commit corresponds to a source code change. Technically, a *Git* commit consists of sections that represent files. Each section consists of chunks that represent a set of lines added or deleted.

*Git*



A commit is considered covered if at least one chunk is covered and we call it *covered bug-fix*. This implies that the original version of these covered chunks was executed by at least one test, but the test did not detect the bug. We consider at least one covered chunk as sufficient to classify the commit as a covered bug-fix, because a single covered chunk implies that the corresponding bug could have been possibly detected by at least one test. This definition can result in a larger set of covered bug-fixes than a manual decision by experts, because our covered bug-fixes may not contain covered parts of the source code which are relevant to the bug.

Covered bug-fix

Analogously, if none of the existing tests executed any part of the original code of the bug-fixing commit, we call it an *uncovered bug-fix*. We use a third category for unclear cases and call a commit an *undecided bug-fix* if we cannot use any of the two previous categories. This occurs in corner cases where a bug-fix introduces new code. For example, lines 1 to 10 in a file are covered by a test and lines 11 to 20 are not covered. A bug-fixing change introduces 5 new lines after the original line 10. It remains unclear whether these new lines should be classified as covered or not.

Uncovered bug-fix

Undecided bug-fix

We ignore file additions, deletions, and moves. If any modification on file level occurs, say a new file is introduced, we expect a change at any other point in the source code reflecting these modifications (e.g., class instantiation, function usage). We only classify this other change and ignore the file modification. There are counterexamples for this reasoning, but we did not find any occurrence in practice. Additionally, the frequency of file moves is less than 1% and the classification is complex, therefore we decided to ignore them for any classification.

Line additions are only classified as covered if the original lines before and after the addition point are covered. We found several counterexamples for a less strict approach that classifies line additions as covered if only one of the stated conditions. For instance, in the case new code is inserted between an uncovered and covered block of code, it remains unclear whether the execution of the new block belongs to the covered block (and the defect could have been detected by a test) or the following uncovered block (where a defect was not detectable by any test execution). For our strict approach, we only expect errors if the source code is badly formatted, e.g., there is no space between two functions and a new function is inserted between or goto is used. We did not find such counterexamples.

Typically, a bug-fixing change also introduces a new (regression) test to verify the absence of this bug. We classify a bug-fix according to coverage before the corresponding regression test is executed and measured in coverage runs. Therefore, such tests do not influence the classification of the related commit. They can influence the classification of future commits, e.g., if a second patch is necessary to fully fix a bug.

We ignore regression tests introduced together with a bug-fix.

### 4.2.3 Empirical Results

This section presents the results of data processing and attempts to answer our main research question on the impact of code coverage. We first discuss the results and implications and then highlight possible threats.

**Data processing.** For the time frame from May 2016 to April 2017, we

Metric	Number
Lines of executable code	Several millions
Number of source files	> 25 000
Full coverage runs	72
Bug-fixing commits	16 215
For bug-fixing commits:	
Diff sections	76 979
Sections covered	24 571
Sections uncovered	15 210
Sections undecided	7 483
Files added	5 595
Lines in new files	1 044 451
Files deleted	1 156
Lines in deleted files	279 674
Files moved	89
Files with content changes	70 139
Lines added in changes	770 325
Lines deleted in changes	471 926
Files with coverage information	47 264
Files without coverage information	22 875
For sections:	
Diff chunks	376 364
Average number of chunks per section	4.89
Average number of chunks per commit	23.21
Skipped chunks source mismatch	4 482
Percentage	0.01 %
Chunks with coverage information	239 119
Chunks covered	94 891
Chunks uncovered	101 907
Chunks undecided	37 839

Table 4.1: Preprocessing statistics for the bug-fixing Git commits collected from May 2016 to April 2017 (1 year).

collected 72 coverage runs and 16 215 bug-fixing commits which represent the same amount of bugs. For these bug-fixing commits, we extracted 76 979 sections from the diff output from Git with cumulatively 376 364 chunks. Among them, 239 119 chunks modify files contained in the coverage data. 4 482 chunks cannot be used because of source mismatch (it remains unclear which file is addressed by the chunk). We found that 94 891 chunks occur in covered parts of the source code, 101 907 chunks occur in uncovered parts of the source code, and for 37 839 chunks it cannot be decided. Based on these results, we identified 24 571 sections as covered, 15 210 sections as uncovered, and 7 483 sections as undecided.

As shown in Table 4.2, cumulatively 8 348 (or 51.48 %) of all bug-fixes occurred in covered parts of the source code, 6 171 (38.06 %) are uncovered, and 1 696 (10.46 %) of all bug-fixes are of type undecided.

Metric	Number	Percentage
Total number of bug-fixes	16 215	100.00 %
Bug-fixes in covered source code	8 348	51.48 %
Bug-fixes in uncovered source code	6 171	38.06 %
Bug-fixes with undecided coverage	1 696	10.46 %

Summary of Table 4.1.

Table 4.2: Numbers of bug-fixing commits (bug-fixes) in our data set by categories defined in Section 4.2.2.4.

**Testedness results.** We compare the observed number  $N_{obs}$  of bugs found in tested code against the *expected* number  $N_{exp}$  of bugs within the covered code. We compute  $N_{exp}$  under the assumption that coverage level is meaningless (null hypothesis), therefore  $N_{exp}$  is the number of all found bugs times the coverage ratio. In our example Fig. 4.2, Scenario 2 has  $N_{obs} = 3$  and  $N_{exp} = 10 \times 0.8 = 8$ . When  $N_{obs}$  is lower than  $N_{exp}$ , we can conclude that the existence of coverage correlates with higher software quality (i.e., fewer bugs) in our test subject.

We approximate the location of bugs in the source code by the location of bug-fixing commits. Furthermore, we conservatively assume that undecided bug-fixes are covered bug-fixes. In other words, we compute  $N_{obs}$  as the sum of covered bug-fixes and undecided bug-fixes.

Contrary to Ahmed et al. [3], we perform this comparison for *each* of the 72 time segments described in Section 4.2.2.1, and not only for a single version of the source code. For each segment we use “local” coverage ratio (and local numbers of covered / uncovered / undecided bug-fixes). For confidentiality reasons, we cannot explicitly state these coverage ratios.

Fig. 4.6 shows a comparison of the numbers of covered bug-fixes plus undecided bug-fixes (sums  $N_{obs,i}$ ) versus the numbers  $N_{exp,i}$  of covered bug-fixes expected under the null hypothesis, for each segment  $i = 0, 1, \dots, 70$ . Only the first 71 segments are shown for presentation reasons. Fig. 4.7 shows the differences  $N_{exp,i} - N_{obs,i}$  in greater detail.

For 70 of 72 time segments, we have  $N_{obs,i} < N_{exp,i}$ , i.e., fewer bugs than expected. The relative reduction of the number of bug-fixes per segment is not large. However, this can be in part attributed to our conservative way of treating undecided bug-fixes as covered bug-fixes.

We also apply the Wilcoxon signed-rank test to reject the null hypothesis that the mean of expected numbers of covered bug-fixes  $N_{exp,i}$  and the mean of numbers of covered plus undecided bug-fixes  $N_{obs,i}$  are equal. The test

Null hypothesis: Coverage has no impact on defect distribution.

We can reject the null-hypothesis.

confirms this with  $p$ -value less than  $2.20 \times 10^{-16}$ , and effect size  $r = 0.612$  which is considered as large ( $r > 0.5$ ).

This non-parametric test is a paired difference test, i.e., we assume that the samples of covered bug-fixes and uncovered bug-fixes are dependent. This is quite likely and indicated by Fig. 4.6. However, we also apply the Wilcoxon-Mann-Whitney-Test which assumes two independent samples. Also here the null hypothesis is rejected ( $p$ -value 0.06) but with small to medium effect size ( $r = 0.13$ ).

*Answer RQ1*  
The number of bugs (represented by bug-fixes) in covered code is smaller than expected if code coverage would have no impact. This holds for 70 out of 72 time segments and is statistically significant for the means.

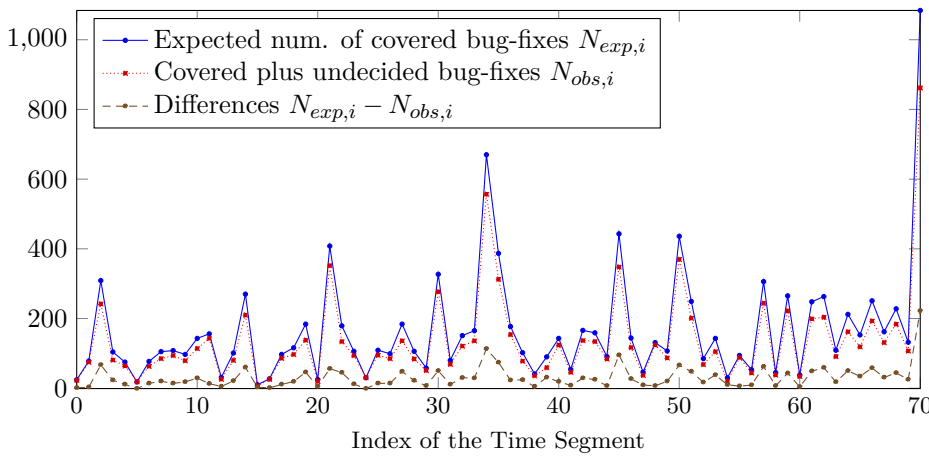


Figure 4.6: Expected numbers of bug-fixes  $N_{exp,i}$ , numbers of covered plus undecided bug-fixes  $N_{obs,i}$ , and their differences for the time segments  $i = 0, \dots, 70$ .

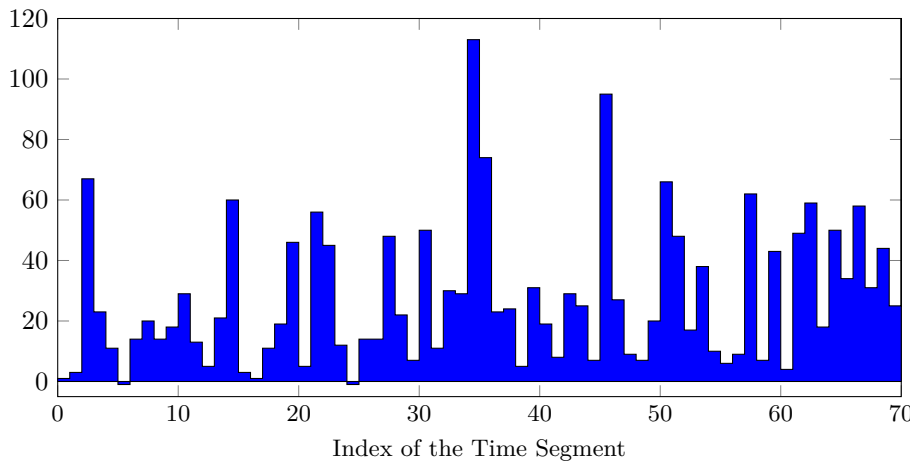


Figure 4.7: Differences  $N_{exp,i} - N_{obs,i}$  for 71 time segments.  $y$ -values  $> 0$  indicate that less bugs occurred than expected.

#### 4.2.4 Discussion of the Results

Our results provide a fairly good support for the thesis that higher coverage levels imply lower levels of future bug-fixes for our test subject. This is in particular strengthened by the fact that in 70 out of 72 time segments

the results agree with the hypothesis. In other words, the positive effect could be observed consistently over time, for large variations of bug numbers. This supports the argument that setting (and reaching) goals for coverage ratio can have a positive effect on the overall quality of our test subject by reducing the number of future bugs.

There are multiple possible explanations for these results. Of course, the most plausible and widely assumed one is that the tests covering code detect bugs often, and since most of these defects are fixed, fewer of them remain.

We also consider an alternative interpretation of the results: that uncovered parts of the source code might include code for which it is difficult to achieve coverage. One well-known example is execution paths in face of rare error conditions, e.g., if `malloc` returns a null pointer. Such return values might not be checked correctly in code because they are not trivial to test, leading to failures if these rare error conditions occur during production usage. A second example is code with a high amount of dependencies. This code requires additional effort to test because all of the dependency interaction must be simulated. A third example comprises code handling external input, e.g., SQL queries. In this case, is it not feasible to test all possible valid and invalid input combinations and fuzzing techniques can require high time and resource costs.

Understanding the relevance of these theories requires further work.

#### 4.2.5 Threats to Validity

Possible threats to our results contain (non-)causality, bad bug classification, wrong coverage granularity and low diversity.

Based on our results, we conclude that fewer bug-fixes occurred in covered parts of the source code than expected. However, we cannot confirm causality, i.e., we cannot conclude that testing caused the reduction of bugs.

We used an existing bug database to retrieve all bugs. These bugs represent only late defects (see Section 4.2.2.2). We argue that these bugs are the more interesting defects because they are more expensive to fix [28], and apparently harder to detect. Analyzing all defects could generate different results for our research question. However, from our experience, it seems impractical to log all early defects during the development, and distinguishing between early and late bugs is hardly possible for young projects.

The bug classification within the bug tracking tool could be wrong. Developers could falsely use the bug category for enhancements or other tasks. The bug tracking tool contains different categories for entries, therefore we do not expect that developers mislabel entries intentionally. But their decision could be wrong. 88% of all entries are bugs, the remaining 12% consists of the categories feature, enhancement, and performance. In addition, SAP enforces a strict policy of bug-labeling. These measures reduce the likelihood of this mislabeling to happen.

We also investigated the possibility that a bug-fixing commit does not only contain the bug-fix, but also unrelated code changes. Herzig et al. found that 15% of all bug-fixes in five open-source projects contain unrelated changes [119]. One of the reasons why such unrelated changes happen is the boy scout rule: “Leave the campground cleaner than you found

Covered code represents code that is tested.

Only bugs for code in version control system.

Bug classification.

Tangled code changes.

it” [187]. According to SAP engineers, unrelated changes rarely happen, because developers focus on fixing the bug and avoid introducing unrelated regressions in the same bug-fixing commit. Enforced commit based code reviews also reduce the probability that unrelated changes are introduced in the same commit as the bug-fixing change. We investigated a subset of 20 bug-fixes and found no changes unrelated to the bug-fix. In the context of our study, such unrelated changes could only increase the number of bugs in covered code due to our conservative classifying approach.

Our coverage data is line-based. A finer granularity (e.g., statement coverage) could produce different results for our research question. However, we would only expect a minor effect.

Coverage from different test types is mixed. The test suites for coverage creation contain a mix of, e.g., integration tests, regression tests, system tests, or performance tests. It remains unclear how this affects the result.

Finally, we only investigate one large industrial system. We do not have access to a second industrial software project with this size, and similar test environment and test data. It is unclear if our results can be reproduced in other large industrial software systems.

Only one study subject.

#### 4.2.6 Conclusions

We applied the binary testedness approach from Ahmed et al. [3] to SAP HANA, a large industrial software project. Instead of using mutants, we used a large set of real bugs and bug-fixing commits. In addition, we introduced multiple data collection points to reduce the risk of losing track of code changes over time. Our results show that a significantly lower number of bugs occur in covered parts of the source code than expected if coverage would be meaningless. For practitioners, our results suggest that setting (and reaching) goals for coverage ratio has a positive effect on the overall quality of our test subject in terms of the amount of future bugs. This confirms previous conclusions from Ahmed et al. [3] and Mockus et al. [194] that show similar effects for coverage.

For SAP engineers, our results confirm the expectation of engineers and management, that *measuring coverage and enforcing coverage goals can be beneficial* to the quality of SAP HANA. We are unable to measure the internal impact of this study, because changes to QA policies require rather complex and lengthy processes.

Measuring coverage and enforcing coverage goals can be beneficial

There are several directions to extend our results in future work. Similar to work of Mockus et al. [194], the binary testedness approach could be used on a component level instead of a system level. A component level analysis could produce comparable results between the components in SAP HANA and could reveal positive or negative correlation factors. Also, a long term experiment over different release cycles of a software project might disclose whether different goals for coverage have an impact on the number of bugs. Mockus et al. [194] found that “there is no indication of diminishing returns (when an additional increase in coverage brings smaller decrease in fault potential)”. It would be interesting to replicate our study on large open-source projects to investigate whether similar findings can be observed for the distribution of bugs and bug-fixing commits.

### 4.3 Granger-Causality between Coverage and Faults

Testing is an integral part of software development to ensure the quality of software as it evolves over time. As the size and complexity of software increase, testing often becomes even more important with developers spending a significant amount of time and resources on it. Coverage is often used by developers to gauge the effectiveness of testing. Developers use coverage to find parts of code not tested that may require their attention. Previous studies analyzed the correlation between coverage and test suite effectiveness by using real bugs as well as mutants, i.e., artificially injected faults. However, these studies only focus on measuring correlations between coverage and bugs (or mutants) instead of studying Granger-causal relationship between them. Essentially, Granger-causality is given if future values of a time series (here number of bugs) can be better predicted using prior values of another time series (here change in coverage percentage). It is argued that in some domains the Granger-causality test is a more reliable indicator of causality than a mere correlation [106]. Also, most of prior studies only consider small open-source projects rather than large industrial systems.

Correlation between coverage and test suite effectiveness

To complement prior work, we analyze SAP HANA and React, an open-source system from Facebook, to uncover Granger-causal relationship between change in coverage and number of future bug fixes found in each system. For each system, we collect a longitudinal dataset spread over one year containing records of code coverage from test runs and bug. We collect data at the file level and use Granger-causality test, which evaluates Granger-causal relationship between two variables, to analyze the relationship between change in coverage and bug-fixes. After filtering constant and non-stationary files, we find that 29% and 33% of the files show Granger-causality between change in coverage and number of bug-fixes for SAP HANA and Facebook React, respectively. For these Granger positive files, we find that the impact of change in coverage on bug fixes affects a majority of the files after a lag of 3 to 4 observation units (i.e., around 2 weeks).

Granger-causal relationship

Complementing prior studies, our study provides added empirical evidence that coverage, as a proxy for testing, has an impact on the number of future bug-fixes. This is the case especially for files exhibiting certain characteristics such as higher lines of code churn and larger amount of commits. Thus, we recommend developers to test these files as they will likely have an impact on the number of bug-fixes.

#### 4.3.1 Introduction

Testing is paramount to ensure a high quality software and is considered an integral part of the software development process. With the increasing size and complexity of software, there is a need to improve testing. More so because inadequate testing can cause millions of dollars [242]. While testing itself does not produce quality, it can help detect faults that can be corrected to improve quality. However, complete testing is often infeasible considering the trade-off between the cost of testing and the number of faults it can detect. As such, developers often make use of measures such as code coverage as a proxy for the adequacy of testing.

In this work, we consider the line coverage ratio, i.e., the percentage of lines executed during testing. Line coverage has been shown to produce better results than more expensive measures like path coverage [103]. Line coverage gives an idea of the adequacy of testing by measuring the number of lines of code executed by the test suite. These results can be used by developers to study the untested parts of the source code and write test cases to potentially reveal undiscovered bugs. Often organizations set testing requirements related to coverage levels and allow committing code only after the code has passed the coverage criteria.

Several related work investigated the question of whether coverage is useful or not. In general, there are two schools of thought on this important yet unsolved question. On the one hand, one group claims that coverage has merit and can lead to an improvement in software quality [194]. This is shown by studies that claim software projects which have higher coverage also have lower bugs in the future. While the other group claims that coverage is a “useless target measure”<sup>6,7</sup> and does not help developers much. Furthermore, some studies fail to come to a conclusion regarding the relationship of coverage and test suite effectiveness [130]. While this topic has received significant attention, studies are mostly concentrated on finding a correlation between coverage and test suite effectiveness. Gopinath et al. analyze hundreds of projects and measure different coverage criteria, such as statement, branch, path and block coverage, to understand which coverage criteria perform the best [103]. They use test cases from the projects as well as generate test cases using Randoop [209]. Inozemtseva et al. analyze five large systems to understand the relationship between coverage, test suite size and test suite effectiveness [130]. A recent study by Ahmed et al. analyzes the correlation between testedness of an element and the number of future bug-fixes using coverage and mutation score [3].

While these studies tackle an important problem, there are several issues:

1. They measure the correlation between various metrics. However, correlation does not imply causation, which measures the impact of one event on the occurrence of an other event (cause and effect). Without studying causation, it is difficult to see the impact of coverage on bugs.
2. The majority of the above studies use mutants, i.e., artificially injected bugs and measure the ability of test suite to kill mutants. However, there are studies that show that mutants may not be representative of real bugs and are sensitive to external threats [104, 200].
3. They use open-source software that might not have rigorous quality assurance processes compared to industry programs. More than 80% of the open-source developers agree that their projects lack testing plans [269].
4. They only consider a snapshot and measure coverage to measure test suite effectiveness. However, testing is a continuous process and measures such as source code, number of test cases, coverage, or number of bugs, evolve over time as the project matures [264]. Furthermore, as per the survey conducted by Runeson [222], newly added functionality should be tested as soon as possible to provide quick feedback to the developers. As such, studying a single version or snapshot may not be an appropriate imitation of the real-world environment as faced by developers.

<sup>6</sup> <http://blog.ploeh.dk/2015/11/16/code-coverage-is-a-useless-target-measure/>

<sup>7</sup> <https://martinfowler.com/bliki/TestCoverage.html>

Limitations of previous work: mutants, only open-source system, single snapshot.



These factors motivate us to perform a longitudinal study where we measure change in coverage and its effect on the number of bug-fixes over a long time period. Our study uses bug-fix instead of mutants.

We investigate a large industrial system SAP HANA to analyze the Granger-causal relationship [106] between changes in coverage and number of bug-fixes. In addition, we complement our analysis with an open-source system Facebook React<sup>8</sup>. Our core hypothesis is that if coverage changes, then it should create an impact on the number of bug fixes — i.e., the number of bug fixes should change too.

<sup>8</sup> See:

<https://facebook.github.io/react/>

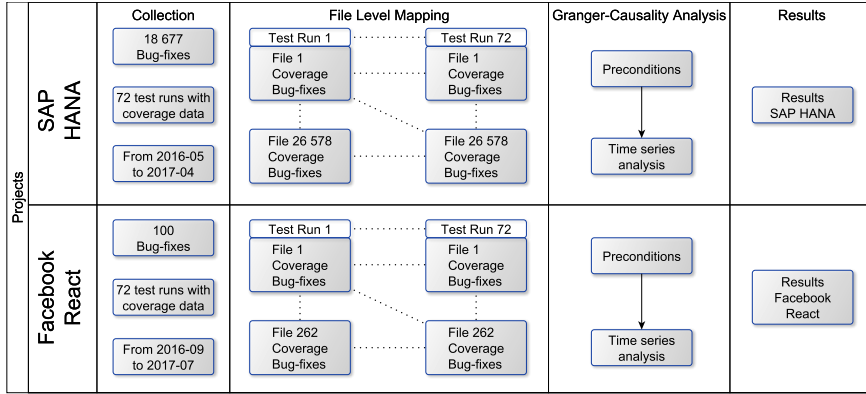


Figure 4.8: Process steps for data collection and evaluation.

To test our hypothesis, we implement the process of data collection and evaluation as shown in Fig. 4.8. We collect over one year (2016-05 - 2017-04 for SAP and 2016-09 - 2017-06 for Facebook) of data consisting of 72 test runs with coverage data and corresponding bug-fixes for each run. We investigate a time frame of one year to reduce the effects of singular events, such as testing phases before releases or development freezes.

Time frame of one year.

We compute the coverage ratio and the number of bug fixes at the file level. For each test run, we first collect all the files in the system and then compute all the lines that are covered by test cases. We measure line coverage and aggregate the values of line coverage over all the lines in a particular file. For each bug-fix, we calculate all the files that are changed. We aggregate the number of bug-fixes that touch a file. In the end, for each file, we have two time series: coverage and bug-fixes collected over the whole observation period. We use the Granger-causality test which measures the causal relationship between two time series [106]. In the Granger-causality test, the hypothesis is to test if a variable  $y$  can be better predicted by using the historical values of  $x$  and  $y$  compared to using the values of  $y$  only. In our case, variable  $x$  is coverage and variable  $y$  is the number of bug-fixes.

The contributions of our work are as follows:

1. The adaptation of Granger-causality test to check whether change in coverage Granger-causes change in number of bug-fixes.
2. Results for a large industrial project with accurate information for 16 000 bugs and bug-fixes over a time frame of one year and results for a large open-source project with 100 bug-fixes over a time frame of one year.

### 4.3.2 Granger-Causality

In this section, we describe the details of Granger-causality [106]. We first start with a definition and discussion of stationary time series as a requirement for the Granger-causality test. We then describe the test itself and conclude with interpretations of possible results.

#### 4.3.2.1 Stationary Time Series

Forecasting techniques for time series often have a precondition, the time series must have a *stationary behavior* [31, 96, 179]. Stationary time series sample values from a stochastic process whose unconditional joint probability distribution does not change when shifted in time. For stationary time series, values such as mean and variance are constant over time. The values of coverage and bug-fixes, when represented as a time series, are often not stationary. This is expected, because the size and complexity of systems typically grows over time as described by Lehman’s Law of software evolution [173]. Often, test suites evolve simultaneously with the change in source code [212]. However, this is similar to using Granger test for its original domain, i.e., to track entities such as the values of prices or inflation that all tend to grow over time.

Stationary behavior

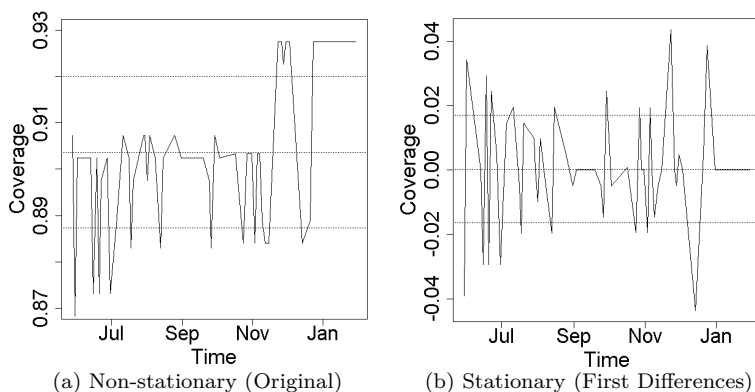


Figure 4.9: Coverage ratio of an example file from SAP HANA over time with non-stationary (original) and stationary (first differences) behavior.

When using time series which are not stationary, one possible approach to make them stationary is to not consider the absolute values of variables, but rather the “first differences” between two consecutive periods [31, 105, 179]. For a time series  $x(t)$ , the first difference is defined as  $x'(t) = x(t) - x(t-1)$ . Fig. 4.9 shows an example of coverage distribution with mean and error bars, i.e.,  $(\mu + \sigma)$  and  $(\mu - \sigma)$  of XYZ file (name withheld for confidentiality) over the entire time period. We apply Augmented Dickey–Fuller test  $adf.test()$ <sup>9</sup> provided by *tseries* package<sup>10</sup> in R<sup>11</sup> to check for stationary behavior. Augmented Dickey–Fuller test (ADF) tests the null hypothesis that a unit root is present in a time series sample. A unit root (also called a unit root process or a difference stationary process) is a stochastic trend in a time series and unit roots can cause unpredictable results in time series analysis.

Fig. 4.9a shows that the coverage value is increasing over time and the distribution shows a non-stationary behavior (p-value > 0.05). To make the distribution stationary, we take the first differences of values  $x(t) - x(t-1)$  and then compute  $adf.test()$ . For  $adf.test()$ , we use the default value of lag<sup>12</sup> as  $nlag = \text{floor}(4 \times (\text{length}(x)/100)^{(2/9)})$ . The second distribution

<sup>9</sup> <https://www.rdocumentation.org/packages/tseries/versions/0.10-42/topics/adf.test>

<sup>10</sup> <https://cran.r-project.org/web/packages/tseries/index.html>

<sup>11</sup> <https://www.r-project.org/>

<sup>12</sup> <https://www.rdocumentation.org/packages/aTSA/versions/3.1.2/topics/adf.test>

(Fig. 4.9b) shows a stationary behavior (p-value  $\leq 0.05$ ). We then use the stationary time series for every file as an input to the Granger-causality test to examine Granger-causal relationship between coverage and bug-fixes.

An analysis of the first difference gains us the same insights with the Grange-causality test as for the original series [31, 105, 179]. Granger and Newbold recommend the first difference approach<sup>13</sup>[105].

#### 4.3.2.2 Granger-Causality Test

The Granger-causality test is a statistical hypothesis test to determine if one time series can be useful in forecasting another ([106]). According to the Granger-causality test, if a signal  $x$  *Granger-causes*  $y$ , then past values of  $x$  should contain information to help predict  $y$ , which is beyond the information contained in the previous values of  $y$  alone. To test the causality between two time series  $x$  and  $y$ , Granger makes use of a statistical test called F-Test to check if  $x$  can help predict  $y$ . In our study,  $x$  is the coverage time series and  $y$  is the bug-fix time series. The Granger-causality test uses bivariate and univariate autoregressive models. A bivariate model includes values from dependent variable  $y$  and independent variable  $x$ , whereas a univariate model only considers prior (lagged) values of variable  $y$ .

For the Granger-causality test, we use the bivariate autoregressive model:

$$y_t = a_0 + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \dots + \alpha_p y_{t-p} + \beta_1 x_{t-1} + \beta_2 x_{t-2} + \dots + \beta_p x_{t-p} + u_t \tag{4.1}$$

and the following univariate autoregressive model:

$$y_t = a_0 + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \dots + \alpha_p y_{t-p} + e_t \tag{4.2}$$

where  $p$  is the autoregressive lag length,  $u_t$  and  $e_t$  are the error terms [160]. The lag value is used as an input parameter to the test. As per [113], the lag value can be estimated via various criteria. The lag length is defined as the number of past values of  $x$  and  $y$  that are considered in the model. Eq. (4.1) and Eq. (4.2) are also called the unrestricted and restricted regression model, respectively. To test whether  $x$  *Granger-causes*  $y$ , the corresponding null hypothesis ( $H_0$ :  $x$  does not *Granger-causes*  $y$ ) is defined as:

$$H_0 : \beta_1 = \beta_2 = \dots = \beta_p = 0. \tag{4.3}$$

In such a case, the lagged values of  $x$  do not add predictive power to the regression equation (Eq. (4.1)). To check this, we compute F-statistic:

$$f = \frac{(RSS_R - RSS_{UR})/p}{RSS_{UR}/(T - k)}, \tag{4.4}$$

where  $k$  is number of explanatory variables in the unrestricted regression model (including the intercept),  $T$  is the number of observations,  $RSS_{UR}$  and  $RSS_R$  are the residual sum of squares, i.e., sum of the squares of the predicted and actual values for the unrestricted and restricted model:

<sup>13</sup> Although they note that they do “not advocate first differencing as a universal sure-fire solution”, therefore we complemented our approach with the ADF test

Can one time series be useful in forecasting another?

Null hypothesis

$$RSS_{UR} = \sum_{t=1}^T u_t^2 \quad RSS_R = \sum_{t=1}^T e_t^2. \quad (4.5)$$

If the value of F-statistic  $f$  is greater than F-critical value at 5% significance level, the null hypothesis is rejected. In this case, the bivariate model is better than the univariate model and  $x$  *Granger-causes*  $y$ .

#### 4.3.2.3 Interpretation of Results

Suppose we have two time series  $T_1$  and  $T_2$ . After carefully checking all preconditions and the application of a Granger-causality test, we can hypothetically obtain the following results:

1.  $T_1$  Granger-causes  $T_2$ .
2.  $T_1$  Granger-causes  $T_2$  for multiple lags. Multiple causalities can exist.
3.  $T_2$  Granger-causes  $T_1$ .
4.  $T_2$  Granger-causes  $T_1$  for multiple lags. Multiple causalities can exist.
5. There is no Granger-causality from  $T_1$  to  $T_2$ .
6. There is no Granger-causality from  $T_2$  to  $T_1$ .

We use the term *unidirectional* if only one direction of causality exists and the term *bidirectional* if both directions show causality. For instance, finding results Item 1 and Item 6 indicates a unidirectional causality. In contrast, finding results Item 1 and Item 3 indicates a bidirectional causality.

The variable lag value ( $p$  in Eq. (4.2)) is responsible for Item 2 and Item 4. In such cases,  $T_1$  and  $T_2$  can have *multiple Granger-causalities*  $c_1$  to  $c_n$ . W.l.o.g.  $c_1$  may have a lag value of 1 and  $c_4$  may have a lag value of 10. In this example,  $c_1$  indicates a short term effect and  $c_4$  may indicate a long term effect. As a practical example, Hu et al. mention in their work [127](Section 1) the work of Freiwald et al. [95] that “revealed the existence of both unidirectional and bidirectional influences between neural groups”. As we see later, our analysis suggests that we also found multiple Granger-causalities (see Table 4.8).

A bidirectional causality can have multiple interpretations and it is important to differentiate such a bidirectional causality from the effects of confounding variables. Maziarz categorizes different interpretations for unidirectional and bidirectional results of Granger-causality analysis [189]. A bidirectional causality may indicate “an instant Granger-causality between the time series”. An *instant Granger-causality* would require a non-existent or minimal lag value. For example, our analysis later shows that for SAP data, lag value 3 has the largest amount of Granger-positive files. In such a case, we can reject the existence of an “instant” Granger-causality.

We may also be able to reject confounding variables if we can reduce the Granger-causality to a real causality. For instance, based on procedures at SAP HANA, bug-fixes influence coverage. SAP developers add new regression tests for fixed bugs. Therefore, two effects occur:

1. A bug-fix modifies source code and therefore may influence coverage.
2. A regression test may execute code that was not tested before and therefore may influence future coverage values.

Unidirectional

Bidirectional

Multiple Granger-causalities

Instant Granger-causality

Hence, we have a causality, and we expect that this causality also results in a Granger-causality of type Item 3. Furthermore, as shown later, there exists also Granger-causality of type Item 2 from coverage to bug-fixes. Conclusively, the causality is bidirectional. Regarding the categorization of Maziarz [189], the bidirectional causality is neither an “instant” Granger-causality, nor a Granger-causality based on the same effect.

We would like to explain such cases of bidirectional Granger-causality with a simplified *analogy in economics*. Let  $T_1$  be the time series (say, yearly) of the gross domestic product (GDP) for country  $C_1$ , and  $T_2$  the time series of the GDP for country  $C_2$ .  $C_1$  and  $C_2$  are neighbors and have a large number of exports and imports between both countries. We further assume that these exchanges positively influence the GDP. In such a situation, we would see a Granger-causality of  $T_1$  for  $T_2$  and a Granger-causality of  $T_2$  for  $T_1$ . Do we observe a bidirectional Granger-causality or two unidirectional Granger-causalities in this case? In our understanding, it is a bidirectional Granger-causality [31, 179]. Furthermore, the differentiation between these two cases is not relevant for the question whether  $T_1$  influences  $T_2$ , because  $T_1$  influences  $T_2$  in both cases. Given these points, we conclude that the presence of unidirectional or bidirectional Granger-causality can both show that a time series  $T_1$  influences a time series  $T_2$ .

Analogy in economics

### 4.3.3 Methodology

In this section, we describe the software artifacts used in this study, and give overview statistics on collected data. We describe our data collection and its preprocessing. Finally, we introduce the process of data filtering and processing required for applying the Granger-causality test.

#### 4.3.3.1 Software Subjects and Data Statistics

We analyze the data of two large software systems. SAP HANA as an industrial system and Facebook React as an open-source system.

**Data statistics:** Table 4.3 gives overview statistics of our data sets. The commit statistics are collected from the version control system Git [14]. For our purpose, we analyze so-called *Git diffs*. A Git diff for a commit consists of  $n \geq 0$  sections. Each section represents a file and contains  $m \geq 0$  chunks. A chunk identifies  $d \geq 0$  deleted lines and  $a \geq 0$  added lines, where  $d + a > 0$ . We analyze the same time period with the same segmentation for both the projects as we aim to have a comparative analysis and see the impact of Granger-causality of coverage on bugs for both projects.

Git diffs

Note that the development of SAP HANA follows a so-called feature branch workflow and a pre-commit review tool is used. Therefore, the pure number of commits is not a suitable activity metric and we have calculated a comparable number for SAP HANA based on other metrics. Consequently, we only provide an approximation.

**Coverage for SAP HANA:** SAP engineers collect line coverage data for all relevant tests (on average) two times a week as discussed in Chapter 3. In the following, we refer to the event of coverage data collection as *test run with coverage analysis*, or simply as a *test run* or *coverage run*. A test run may consist of several coverage files that we aggregate to a single file.

Coverage run

Metric	SAP HANA	Facebook React
Test runs with coverage analysis	72	72
Number of source files	26,578	262
Bugs	16,757	100
Bug-fixing commits	18,677	100
Total commits	≈ 210,000	1,606
New files added	5,595	572
Lines added in chunks	1,044,451	138,502
Lines deleted in chunks	471,926	132,849
Observation Period	May'16 - April'17	Sep'16 - Jun'17

Table 4.3: Statistics on the coverage data and bug-fixes collected for SAP HANA and Facebook React.

In this study, we use the *coverage ratio* of a file, that is, the number of executed lines over the number of all executable lines in a file. For example, a file with 100 lines and 20 executed lines has a coverage ratio of 0.20.

We also experimented with an alternative (and abandoned) file coverage metric where a file was considered as covered if any line inside the file was hit. We found that nearly all C++ source files were marked as hit because the compiler generates executable code for each namespace that is executed by all tests independent of the source file content. The majority of files contain namespace because in a large C++ project as SAP HANA, namespaces are necessary to organize and structure all functionality.

SAP measures the coverage for each test suite. This study focus on coverage for the whole project, we did not investigate coverage per test suite or coverage per test. Therefore, the line-based coverage of a file can contain the coverage of multiple tests and multiple test suites. For instance, if source code lines 1 and 2 are covered by test  $t_a$ , and lines 3 and 4 are covered by test  $t_b$ , then we only measure that lines 1,2,3, and 4 are covered.

**Defects for SAP HANA:** We classify defects by the time of their detection into *early detected* and *late detected*. Our study focuses on the latter type of defects, which tend to have a higher cost impact [28, 226]. In the following, we use the term *bug* to refer to such late detected defects.

We classify a defect as early detected if it is found by a developer before a commit, or found by any quality assurance activity before a commit is merged into the main branch. Such activities include local test runs, peer reviews, and automatic test runs within the continuous integration process.

Defects detected late are discovered at later QA steps, or might even escape the in-house testing. Examples of such later QA steps are additional integration testing, component tests, manual testing, fault discovery during internal usage of SAP HANA (“self-hosting”), or fuzz testing.

**Facebook React:** Facebook React is an open-source JavaScript library to build user interfaces. React is managed by Facebook and is hosted on GitHub<sup>14</sup>. According to the GitHub page, React is a “declarative, efficient, and flexible JavaScript library for building user interfaces”.

**Coverage for React:** Coverage data for React is openly available on Coveralls.io<sup>15</sup>. Coveralls.io provides several features for a repository such as total percentage of coverage, individual file coverage, line coverage, coverage trend over time, change in coverage due to commits, integration with GitHub, and various continuous integration services.

Coverage ratio

Early and late detected defects.

<sup>14</sup> Full project details can be found at <https://github.com/facebook/react>

<sup>15</sup> See for details <https://coveralls.io/github/facebook/react>

**Defects for React:** As React is hosted on GitHub, we use Git logs to obtain the history of commits and then filter out commits that are related to bug-fixes based on keyword analysis. This keyword-based approach results in a less accurate data set than our data set for SAP HANA. However, the approach is commonly used in related work [231].

#### 4.3.3.2 Data Processing

In this subsection, we describe our data collection procedures.

**Coverage data for SAP HANA:** Our study comprises coverage data over one year (from May 2016 until April 2017) with results of 72 valid test runs with coverage analysis. Chapter 3 describes the format of coverage data and the processing details.

**Bug Data for SAP HANA:** Each of the bugs (i.e., late detected defects as defined in Section 4.3.3.1) is stored in a bug tracking tool. For our study, we link each bug record to a *Git* commit. This can be done in both directions: each bug record contains a reference to the commit with the bug-fix (typically this reference is a *Git* commit hash). Conversely, a separate database contains for each bug-fixing commit a reference to a corresponding bug record. Consequently, we know for each bug (i.e., bug record) its bug-fixing commit, and vice versa. SAP engineers have collected, maintained, and used this information for many years within the HANA project, and so we can assume that the results are reliable. The above-described data allows generating, for each bug, the list of files changed by the corresponding bug-fixing commit, including the list of lines changed in each file.

Note that for each bug there can be multiple bug-fixing commits, and vice versa. However, for the purpose of this study, we assume a one-to-one relationship and ignore additional relations if they exist.

Reliable data provided by a bug tracking tool of SAP.

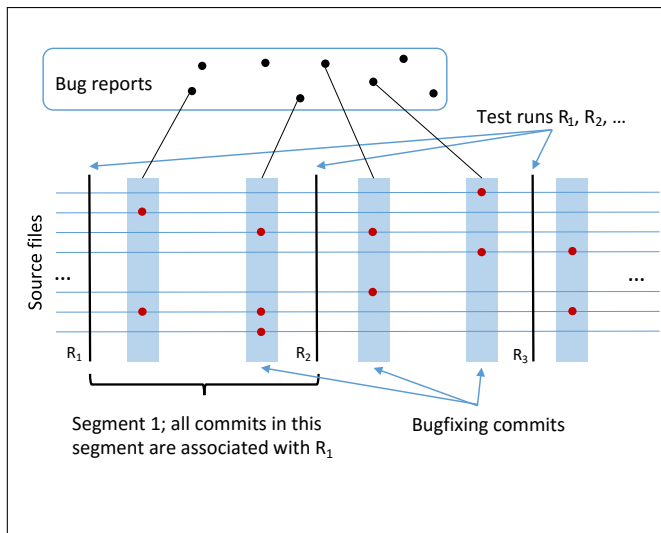


Figure 4.10: Time segments, test runs, and bug-fixing commits.

**Moving time window for SAP HANA:** Intermediate code changes between a test run with coverage and a bug-fixing commit  $c$  prevent us from directly applying the coverage information to  $c$ . Previous work by Ahmet et al. [3] proposed to solve this problem with elaborated change tracking approaches like a *GumTree Differencing Algorithm* [87]. We propose a

GumTree Differencing Algorithm

simpler approach resembling a *moving window analysis* on time series as shown by Fig. 4.11. We expect that our approach allows more accurate mapping of bug-fixes to coverage data compared to the GumTree approach at the cost of considerable higher analysis effort. In addition, we are able to deploy analysis techniques for time series.

Moving window analysis

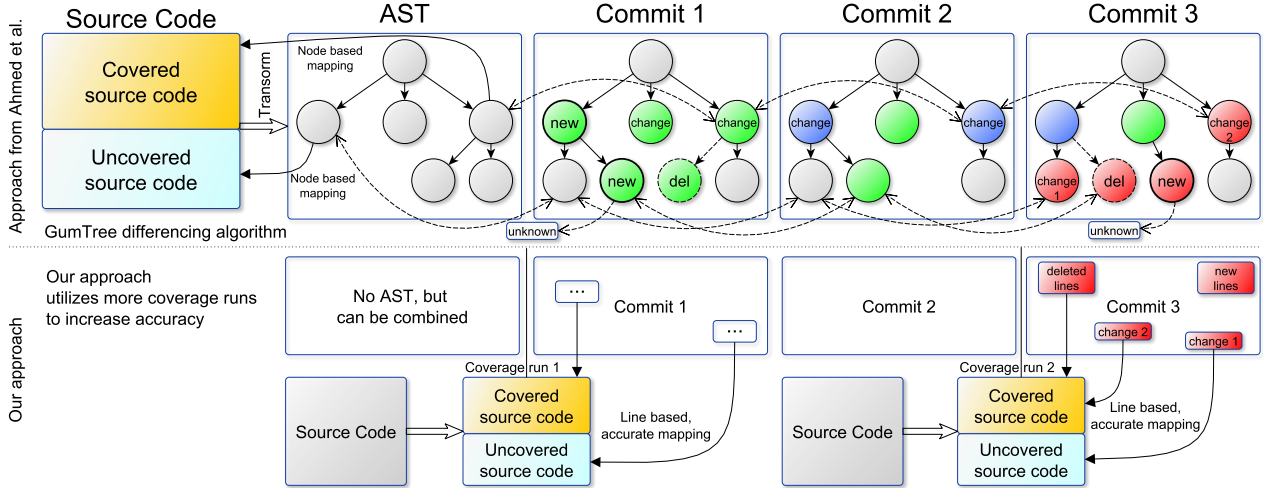


Figure 4.11: The GumTree [87] approach tracks the nodes of an abstract syntax tree (AST). This can reduce accuracy for consecutive changes. For commit 3 in this example, change 1 is accurately mapped as covered. Change 2 is unclear due to intermediate deleted and inserted nodes. The new node and the deleted node in commit 3 cannot be mapped. Our approach can accurately map these examples.

For our moving time window approach, we split our data into consecutive time segments, and measure the line coverage and the number of bug-fixing commits separately for each of these segments. In detail, we divide our data into time segments with boundaries determined by the test runs with coverage analysis. Within each such a segment, each bug-fixing commit is associated with the closest preceding test run. To illustrate how we divide the data into time segments, we have plotted Fig. 4.10. To read this figure, consider  $R_1$  as the first test run that outputs coverage information for each source code file in the system. We then calculate the *number of bug-fixes* made between  $R_1$  and the next test run  $R_2$  by using the timestamps of test runs ( $R_1$  and  $R_2$ ) and comparing them with the timestamps of bug-fixes from the log files. In Fig. 4.10, both bug-fixing commits in the Segment 1 are associated with the test run  $R_1$ .

For each of these bug-fixes, we have information about the files that are changed and the corresponding lines added, deleted or modified. We aggregate the number of times a file is changed by counting different bug-fix commits. Thus, for each test run  $R_i$  and for each source code file, we have corresponding coverage value and the number of bug-fixes applied on that file. We repeat this procedure for all the test runs in our dataset. Our analysis considers only the “local” relations within a single segment, i.e., relations between a test run and (only) the associated bug-fixing commits within the same segment. Due to the limited time span of a segment, the risk of encountering a code change between a test run and a bug-fixing commit is low. Even if the relations test run vs. commits are considered only per segment, the Granger-causality test incorporates data over all segments, which allows us analyzing data over arbitrary time spans.



In comparison to the GumTree approach, our approach reduces the chances of conflicting intermediate commits between a test run with coverage analysis and a bug-fixing commit. Such conflicting intermediate commits can result in an inaccurate mapping of the bug-fixing commit to coverage data. We highlight two examples in Fig. 4.11. The new node in commit 3 cannot be mapped to the original coverage because there is no AST information available. Change 2 in commit 3 may be unclear because of the intermediate commits. Our approach, in this example, accurately maps all changes to the most recent coverage data.

Moving time windows is more accurate.

**Coverage Data for React:** To select builds to be analyzed, we first extract those builds on “master” branch that were done between September 2016 to June 2017 from Coveralls.io. In addition to general information of the builds, we also extract information of individual files in each build (both changed and unchanged) from the respective builds’ detailed report locations in Coveralls.io. The time period of September 2016 – June 2017 was chosen since the entries in this period have a relatively stable reported number of files, as well as small intervals between entries (typically multiple builds per day). We subsequently prune build reports with anomalous numbers of reported files; for example, an entry with 0 reported files between a series of entries reporting 246 files each. The rationale here is that they are likely to result from wrong configuration or other errors. After this pruning, we find 1563 builds with numbers of files ranging between 219 and 262.

**Bug Data for React:** We identify bug-fixing commits by applying keyword search on the commit log messages using a set of keywords (“error”, “bug”, “fix”, “issue”, “mistake”, “incorrect”, “fault”, “defect”, “flaw”) that have shown to achieve high accuracy [217, 218]. We associate each bug-fixing commit with its nearest earlier build, using the commit file list to create counts at file level granularity.

**Moving time window for React:** See description for SAP HANA.

#### 4.3.3.3 Applying the Granger-Causality Test

To apply the Granger-causality test we need to check if future values of coverage and bug fixes depend on their previous values as Granger-causality can be applied to stationary multivariate autoregressive processes. To check this, we compute *autocorrelation*, i.e., the correlation between current and previous values. We calculate Spearman’s correlation for all the lag values 1 to 10, i.e., the correlation between  $i$  and  $i + 1$ ,  $i$  and  $i + 2$  and so on. For SAP HANA, we find a strong correlation for both coverage and bug fixes with Spearman’s correlation varying from 0.94 to 0.97 for coverage and 0.85 to 0.98 for bug-fixes, all with significant p-values. For Facebook React, we find a strong correlation for coverage ranging from 0.93 to 0.99, and a weak correlation for bug fixes 0.07 to 0.13. However, the p-values are significant for both coverage and bug fixes. Thus, we find that future values of coverage and bug fixes depend linearly on their past values.

Autocorrelation

To apply the Granger-causality test to find out causal relationships between the time series of coverage and the number of bug-fixes, we need to check for several preconditions. For each file  $f$ , the *coverage time series*  $Coverage[f]$  is a sequence of coverage values for  $f$  over the consecutive time

Check for several preconditions  
Coverage time series

segments. Similarly, the *bug fix time series* Bug-fixes[f] is a sequence of numbers of bug fixes related to f over the consecutive time segments. We have a coverage time series and a bug-fix time series for each file. Note that in Eq. (4.1), Coverage[f] corresponds to the independent variable  $x$  and Bug-Fixes[f] corresponds to the dependent variable  $y$ . Algorithm 1 describes the steps to check for the fulfillment of preconditions and run *grangertest*. In this algorithm,  $F_i$  is the set of all input files of the system, and  $cov\_vec(t)$ ,  $bug\_vec(t)$  denote the  $t^{th}$  element of a respective time series. We apply  $c\_check_1$  (line 3) to check the following preconditions:

- P1: The coverage time series for a file must be present 30% of the whole observation period. A time series that is present only for a short period of time is called dayfly classes and might not be useful for prediction [170]. We need a considerable history of coverage values to help detect Granger-causal relationship with bug-fixes. We discard files that do not satisfy this criterion. Previous studies also ignore such classes [54, 55].
- P2: The coverage time series must not be constant. We found that for some files the values of coverage do not change over the whole observation period. As changes to independent variables are important to observe variations in the dependent variable for the Granger-causality test, we discard files for which coverage remains the same.

Bug fix time series

Coverage time series must be present at least 30%.

Coverage time series must not be constant

---

**Algorithm 1:** The Granger-causality test.
 

---

**Input** :  $F_i$ : Set of input filesn: Lag value for *grangertest***Output** :  $F_o$ : Set of output files for which Granger-causal relationship is observed

```

1 for all  $f \in F_i$  do
2   cov_vec = Coverage[f];
3   if  $c\_check_1(cov\_vec)$  then
4     cov_vec $_{fd}(t) = cov\_vec(t) - cov\_vec(t - 1)$ 
5     if  $c\_check_2(cov\_vec_{fd}(t))$  then
6       bug_vec = BugFixes[f];
7       bug_vec $_{fd}(t) = bug\_vec(t) - bug\_vec(t - 1)$ 
8       if  $b\_check(bug\_vec_{fd}(t))$  then
9         |  $grangertest(bug\_vec_{fd}(t) \sim cov\_vec_{fd}(t), order = n)$ 
10      end
11     end
12   end
13 end

```

---

For every file  $f$  that pass  $P1$  and  $P2$ , we compute the first differences  $cov\_vec_{fd}(t)$ , i.e., the difference between coverage values at time  $t$  and  $t - 1$  (line 4). Then, we apply  $c\_check_2$  (line 5) to check precondition  $P3$ :

- P3: The coverage time series must be stationary (see Section 4.3.2.1), which is a precondition for applying the Granger-causality test. To test for stationary behavior, we apply  $adf.test(vec)$  provided by *tseries* package

Coverage time series must be stationary

in R, where  $vec$  is a time series. The null hypothesis is that time series is non-stationary whereas the alternate hypothesis is that time series is stationary. The function implements *Augmented Dickey-Fuller* test to check for stationary behavior [96].

For every file  $f$  that satisfies  $P1$ ,  $P2$  and  $P3$ , we obtain its corresponding bug-fix time series (i.e.,  $BugFixes[f]$ ). Similar to coverage time series, we compute first differences,  $bug\_vec_{fd}(t) = bug\_vec(t) - bug\_vec(t - 1)$  (line 7) for the bug time series. We then apply `b_check` (line 8) to ensure that the following pre-condition is met:

- $P4$ : Similar to  $P3$ , bug-fix time series must be stationary. The null hypothesis is that time series is non-stationary whereas the alternate hypothesis is that time series is stationary. We apply `adf.test(vec)`, where  $vec$  is a time series of bug-fixes, and discard files that are non-stationary.

After the above checks, the time series  $cov\_vec_{fd}(t)$  (coverage) and  $bug\_vec_{fd}(t)$  (bug-fixes) pass preconditions  $P1$  to  $P4$ . We then apply `grangertest`<sup>16</sup> provided by `lmtest`<sup>17</sup> package in R:

$$grangertest(bug\_vec_{fd}(t) \sim cov\_vec_{fd}(t), order = n) \quad (4.6)$$

where  $order$  is the lag value that needs to be specified as an input parameter. Lag value is defined as the number of past values of the independent variable that will be considered to predict the current value of the dependent variable. We run the Granger-causality test with order value from 1 to 10 and take the union of files that return significant results for the test. Similar to related work [56], we assume here that for a file to be Granger positive it is sufficient that it passes the test at any of these lag values. We consider files as *Granger positive* and *Granger negative* if they show significant and insignificant results for the Granger-causality test, respectively.

Bug-fix time series must be stationary

<sup>16</sup> <https://www.rdocumentation.org/packages/lmtest/versions/0.9-35/topics/grangertest>

<sup>17</sup> <https://cran.r-project.org/web/packages/lmtest/index.html>

Granger positive  
Granger negative

Preconditions	SAP HANA		Facebook React	
	%	Files	%	Files
Start	100.00	26 578	100.00	562
P1	96.20	25 567	47.51	267
P1+P2	56.26	14 952	43.06	242
P1+P2+P3	51.30	13 635	21.53	121
P1+P2+P3+P4	20.92	6 428	9.43	53

Table 4.4: Percentage and absolute number of files that satisfy preconditions  $P1$ ,  $P2$ ,  $P3$  and  $P4$ .

Table 4.4 shows the percentage and the absolute number of files that satisfy  $P1$  to  $P4$ . Initially, there are 26 578 files for SAP HANA. After filtering out files that are present in at least 30% of the time series ( $P1$ ), we have 25 567 files, or 96.20 % of the original data. We then perform a stationarity test on both the coverage and bug-fixes time series and also filter out files with no changes in the coverage value over the whole period ( $P3$ ,  $P4$ , and  $P2$ , respectively). In the end, 5 560 files satisfy all criteria. These files are then used to run the Granger-causality test.

For Facebook React, we have 562 files initially. We apply the same steps as for SAP HANA and get 53 files that to run the Granger-causality test.

#### 4.3.4 Findings

In this section, we answer the following research questions (RQ) for our two study subjects, SAP HANA and Facebook React:

- RQ2 How many files show a Granger-causal relationships between time series with coverage and bug-fixing commits (BFC)?
- RQ3 How many files show a Granger-causal relationships between time series with coverage and non-bug-fixing commits (NBFC)?
- RQ4 How does the lag value affect the results?
- RQ5 What are differences between files with positive and negative results for the Granger-causality test?

##### 4.3.4.1 RQ2 Granger-Causality Between Coverage and BFC

**Motivation:** We want to analyze the causal relationship between coverage and the number of bug-fixes. This tells us if coverage has an impact on the number of bug-fixes, i.e., whether covered program elements show a higher or lower number of bug-fixes. Our hypothesis is that a program element with changes in coverage over time should also show changes in number of bug-fixes over time, i.e., in number of bugs over time.

**Methodology:** After filtering out files which do not satisfy the stationarity test, we perform the Granger-causality test in R using *grangertest* function (see Section 4.3.2.2). We perform the Granger-causality test for every file, i.e., for every time series of coverage and bug-fixes.

Category	SAP HANA	Facebook React
Number of Files	6 428	53
GPF <sub>1-10</sub>	1 893	18
Bug-Fixes	18 677	100
Bug-Fixes <sub>GPF</sub>	5 866	97
No. of Files/Bug-Fixes	34.42 %	53.00 %
GPF/Bug-Fixes <sub>GPF</sub>	32.27 %	18.56 %

**Findings:** Table 4.5 shows the results of running the Granger-causality test on our dataset. We find that 1 893 files (29.45 %) return significant results for the Granger-causality test for SAP HANA. Similarly, for Facebook React 33.96 % of the files (18/53) return significant results.

We use a significance level of 95 % ( $\alpha \leq 0.05$ ) for the Granger-causality test. Table 4.6 shows the number of Granger positive results at intervals of the significance level separated by 1 %. We observe that more than 68 % of the 1 893 files have p-value less than 0.01 for SAP HANA and more than 88 % of the files have p-value less than 0.01 for Facebook React.

##### Answer RQ2

29.45 % (1 893/6 428) and 33.96 % (18/53) of the files show positive result for the Granger-causality test for SAP HANA and Facebook React, respectively. 68.83 % and 88.88 % of all files with significant results have p-value less than 0.01 for SAP HANA and Facebook React, respectively.

Table 4.5: The results of running the Granger-causality tests for both applications. The row names denote: No. of Files - Total number of files that are used for the Granger-causality test, GPF (Granger Positive Files) - Files that return significant result for the Granger-causality test, Bug-Fixes - Total number of bug-fixes, Bug-Fixes<sub>GPF</sub> - Total number of bug-fixes applied to files that satisfy the Granger-causality test.

Significance Range	Number of Files	
	SAP HANA	Facebook React
$0.04 < p - val \leq 0.05$	97	0
$0.03 < p - val \leq 0.04$	140	1
$0.02 < p - val \leq 0.03$	142	0
$0.01 < p - val \leq 0.02$	211	1
$0.00 < p - val \leq 0.01$	1 303	16

Table 4.6: Granger positive p-values for files.

#### 4.3.4.2 RQ3 Granger-Causality Between Coverage and NBFC

**Motivation:** The answer to this research question provides insights into the significance of the results for RQ2. There might exist a general Granger-causality between coverage and (all types of) commits. In this case, there would exist a similar Granger-causalities for BFC and NBFC. To investigate this possible scenario, we perform additional experiments and test two hypotheses for bug-fixing and non-bug-fixing commits:

1. There is no difference in the number of Granger positive files between bug-fixing and non-bug-fixing commits *or* there are more Granger positive files for non-bug-fixing commits than for bug-fixing commits.
2. For a Granger positive file  $f$  such that the difference of  $f$ 's line coverage in between test runs Granger-causes a difference in the number of  $f$ 's bug fixing commits in between test runs, the difference of  $f$ 's line coverage in between test runs also Granger-causes a difference in the number of  $f$ 's non-bug-fixing commits in between test runs.

**Methodology:** Compared to our methodology for bug-fixing commits, we must now filter for non-bug-fixing commits. However, this results in a different population for our experiment as these commits vary by several characteristics. Therefore, we enumerate each issue and state how we control for these variations in our methodology.

- The statistics for number of files changed per commit is different for non-bug-fixing-commits (median 2, mean 7.52) and bug-fixing commits (median 2, mean 5.02). We control this variable by selecting non-bug-fixing commits in such a way that the mean and median are similar to bug-fixing commits. Otherwise, we would have more candidates for Granger-causality which would impact the results.
- A non-bug-fixing commit can include non-relevant files such as files that are not required for the product SAP HANA. We control this variable by selecting only commits that change files contained in the set of all files contained in coverage results. Otherwise, unrelated files, such as a documentation files or sources files for other products, would impact the results because there cannot be any Granger-causality for such files.
- The number of non-bug-fixing commits is larger than the number of bug-fixing commits. We control this variable by selecting (randomly) the same number of non-bug-fixing commits as we found bug-fixing commits. Otherwise, the population would be different for both experiments.

For the analysis of the bug-fixing commits in Facebook React, we use all non-bug-fixing commits due to the small number of commits.

For both SAP HANA and Facebook React, we determine the number of common files by calculating the intersection between the set of files that show Granger-causality for bug-fixing commits and the set of files that show Granger-causality for non-bug-fixing commits.

**Findings:** From Table 4.7, we find for SAP HANA 697 GC-positive files for non-bug-fixing commits compared to 1893 GC-positive files for bug-fixing commits. Therefore, we can reject the null hypothesis 1 for SAP HANA as there is a factor of 2.7 less GC-positive files for non-bug-fixing commits. For Facebook React, we find 57 GC-positive files for non-bug-fixing commits compared to 18 GC-positive files for bug-fixing commits. Therefore, we can not reject the null hypothesis 1 for Facebook React. The small sample size for Facebook React leads to inconclusive results, i.e., we can neither reject the null hypothesis nor accept the null hypothesis.

	Granger Positive Files	
	Bug-Fixing Commits	Non-Bug-Fixing Commits
SAP HANA	1 893	697
Facebook React	18	57

Table 4.7: Granger positive files for bug-fixing and non-bug-fixing commits.

We found that for SAP HANA and Facebook React, 319 files and 13 files are common between bug-fixing commits and non-bug-fixing commits that are GC-positive, respectively. This shows that the percentages of files that are GC-positive for both experiments (bug-fixing/non-bug-fixing commits) are rather low (17%/46% for SAP HANA, 72%/23% for React). We conclude that we find different files that are GC-positive for each experiment.

Given these numbers, for SAP HANA, in 1 574 files, we can reject the null hypothesis 2 and in 319 cases, we cannot reject the null hypothesis 2. We did not further investigate whether the 319 cases contain cases where the lag value for a file in the bug-fixing analysis is different from the lag value of a file in the non-bug-fixing analysis. We would expect that such differences exist and therefore the amount of files with identical results is lower. However, we did not calculate the exact number because the comparison on this level is rather complex (a file can have multiple lag values) and provides limited insights. For Facebook React, in 5 files we can reject the null hypothesis 2 and in 13 files we cannot reject the null hypothesis 2. We argue that the number of cases is too small to draw any conclusion. We speculate that the lower number of cases where we can reject the null-hypothesis happens because of the low amount of files and the different characteristics for Facebook React compared to SAP HANA as shown by Table 4.9. In addition, for React, we use a heuristic to detect bug-fixing commits and such bug-fixing commits may contain changes unrelated to the bug-fix. The analysis for SAP HANA does not share these issues.

*Answer RQ3*

For SAP HANA, there are a factor 3 more Granger positive files for bug-fixing commits than for non-bug-fixing commits. Furthermore, for the majority of the cases in SAP HANA and Facebook React, files that show Granger-causality on bug-fixing commits are different from files that show Granger-causality on non-bug-fixing commits.

#### 4.3.4.3 RQ4 Analysis of Lag Values

**Motivation:** The Granger-causality test is sensitive to the lag selection. The lag corresponds to a delay, which is unknown before. Due to the practical and large scale development process at SAP, it is plausible that different delays occur. As such, we want to investigate which value of lag can return the most positive results. This will be helpful for developers to know the amount of time that is likely to pass before they observe Granger-causality between coverage and bug-fixes.

**Methodology:** Originally, we fixed a single value of lag. In this experiment, we change the value of lag from 1 to 10 for each pair of series of coverage and bug-fixes. If one of the lag returns a significant result ( $p \leq 0.05$ ), we consider this case as existence of Granger-causality.

Lag	SAP HANA		Facebook React	
	Files	Files(%)	Files	Files(%)
1	583	30.80	7	13.21
2	749	39.57	11	20.75
3	779	41.15	14	26.42
4	774	40.89	9	16.98
5	689	36.40	6	11.32
6	684	36.13	5	9.43
7	705	37.24	5	9.43
8	708	37.40	5	9.43
9	712	37.61	5	9.43
10	731	38.62	7	13.21

Table 4.8: Granger positive files over different lag values for both projects. Highest values are highlighted.

**Findings:** Table 4.8 shows the lag values and the number of Granger positive files for each lag value and each study subject. For each lag value, there are certain number of files that show Granger-causality. For example in SAP HANA, for lag value 1, 583 (out of 6,428) files show Granger-causality, whereas 749 files show causal relationship for lag value 2. When multiple lags show Granger-causality for a particular file, we take the one with the lowest p-value. For SAP HANA, we observe that lag value 3 returns 779 files, which is the highest number of files with Granger-causality among all the lag values. Similarly, for Facebook React, lag value 3 returns the highest number of files with Granger-causality. Furthermore, there is not a substantial change in the number of Granger positive files between lag values 6 and 9 for both SAP HANA and Facebook React. The results for lag value 10 show a small increase in number of Granger positive files for both SAP HANA and Facebook React. However, the small number of files for Facebook React may lead to artificial and insignificant results.

#### Answer RQ4

The number of files that show Granger-causality varies across different lag values. For both SAP HANA and Facebook React, the lag value 3 shows the highest number of Granger positive files with 41.20% and 26.42% of the files that exhibit a Granger-causal relationship. There is no significant change in the number Granger positive files between lag value 6 and lag value 9 for both the projects.

#### 4.3.4.4 RQ5 Characteristics of Granger Positive Files

**Motivation:** We want to understand the characteristics of the files for which Granger-causality is observed in comparison to those files for which Granger-causality is not observed.

**Methodology:** For SAP HANA, we compare the 1 893 files for which we observed Granger-causality against the 4 535 files that satisfy pre-conditions for the Granger-causality test but for which Granger-causality is not observed. Similarly, for Facebook React, we compare 18 files with Granger-causality against 35 files for which Granger-causality is not observed. For each file, we compute the LOC churn, i.e., the number of lines added and deleted over the time series. This estimation will help us understand the number of changes that are made to these files over 1 year period and over many successive test runs. Further, we compute all the number of commits made to each file and the number of developers who have contributed to each file during the whole observation period. We run Mann-Whitney Wilcoxon one-tailed test [185] to compare the mean values of different metrics for Granger positive and negative files. We use one-tailed test to determine if the difference between the two groups is in a specific direction, whereas two-tailed test is used to determine if there is any difference without giving any specific direction as to which is bigger or smaller.

**Findings:** The null hypothesis is that there is no difference in LOC churn, i.e., the number of lines added and deleted, for Granger positive files and negative files. The alternative hypothesis is that Granger positive files have higher LOC churn than Granger negative files. The mean values of LOC churn for Granger positive and negative files for SAP HANA are 472.41 and 442.64, respectively and the median values are 111 and 96, respectively. We observe that p-value is 0.004 which shows that Granger positive files have significantly higher LOC churn than Granger negative files. For Facebook React, while the mean and median values are higher for Granger positive files than Granger negative files, we do not observe a significant difference between these two groups. This is possibly due to a limited number of data points for Facebook React.

Similar to above, we compute the number of commits affecting a file over the entire time period and compare Granger positive and Granger negative files. The null hypothesis is that there is no difference in the number of file changes for Granger positive and Granger negative files, whereas the alternative hypothesis is that Granger positive files have a higher number of file changes than Granger negative files. Using Mann-Whitney Wilcoxon one-tailed test, we observe that there is a significant difference (p-value = 0.003), i.e., Granger positive files have a significantly higher number of changes than Granger negative files for SAP HANA. For SAP HANA, the mean and median values for changes in files with significant Granger-causality are 12.69 and 7, whereas the corresponding values for Granger negative files are 12.88 and 6, respectively. For Facebook React, the mean and median for Granger positive files are 32.33 and 18, whereas those for Granger negative files are 20.77 and 9. Different from the results we observed for SAP HANA, we do not observe a significant difference between the two groups for Mann-Whitney Wilcoxon test for Facebook React.



Furthermore, we analyze the difference between the mean values for the number of developers who have contributed to Granger positive files against those who have contributed to Granger negative files. For SAP HANA, we do not observe a significant difference in the number of developers between Granger positive and Granger negative files. The mean values for these files are 4.58 and 4.67, respectively, with a median value of 3 for both groups. For Facebook React, while the values of mean and median are higher for Granger positive files over Granger negative files, the difference is insignificant. This is probably due to the limited number of data points for Facebook React. Table 4.9 shows the mean and median values for different metrics for Granger positive and negative files for SAP HANA and Facebook React, respectively.

Metric	Granger Positive		Granger Negative	
	Mean	Median	Mean	Median
SAP HANA				
LOC difference*	472.41	111	442.64	96
Number of commits*	12.69	7	12.88	6
Number of developers	4.58	3	4.67	3
Facebook React				
LOC difference	168.17	29.50	134.40	29
Number of commits	32.33	18	20.77	9
Number of developers	8.06	6.50	6.74	6

Table 4.9: Mean and median values of different metrics for Granger positive and negative files for SAP HANA and Facebook React. A \* indicates  $p < 0.05$ . The lack of statistical significance for React may be due to limited data points.

Thus, the result for SAP HANA shows that coverage Granger-causes bugs. The result for Facebook React is inconclusive (i.e., it is not statistically significant). The two results are not contradictory. We cannot ascertain whether the results for React support or refute the relationship observed for SAP. The size of the React dataset could cause the inconclusive result.

*Answer RQ5*

For SAP HANA, files that show positive results for the Granger-causality test exhibit a significant difference from files that show Granger negative results in terms of LOC churn and number of commits affecting them. For Facebook React, we do not observe a significant difference between Granger positive files and Granger negative files, possibly due to a limited number of data points for Facebook React.

#### 4.3.5 Discussion

In this section, we discuss our findings and their implications. We also state and discuss possible threats to the validity of our work.

##### 4.3.5.1 Implications

In this study, we highlight the Granger-causality between coverage and bug-fixes. We find Granger-causal relationship for over 29% of the files for SAP HANA and over 33% of the files for Facebook React. Based on the categorization in Section 4.3.2.3, these are of type Item 2. This shows that coverage has an impact on bug-fixes but only for certain files.

Not all files show Granger-causal relationship. However, we argue that this is expected because not all files are modified equally over time. In our experiment, we analyze the Granger-causality of coverage changes over time (time series  $T_1$ ) to bug fixes (time series  $T_2$ ) on file level. Due to the file level context, we have in fact multiple time series  $T_{1all} = \{T_{11}, \dots, T_{1n}\}$  where  $n$  is the number of files. Therefore, our experiment consists of, in fact,  $n$  Granger-causality measurements. We do not expect, and it is in fact not possible, that all time series in  $T_{1all}$  show Granger-causality. There are multiple time series in  $T_{1all}$  where no bug fix occurs and, therefore, we cannot conclude any information about the changes or Granger-causality (that acts on changes) for such time series. In addition, based on an intuitive understanding of the appearance of bugs, not all bugs are found by tests and not all places and files have the same probability to contain bugs. Therefore, it is expected that not all time series in  $T_{1all}$  show Granger-causality.

One time series for each file.

Our results show Granger-causality for some time series in  $T_{1all}$ . The importance of our result is not the percentage of time series that show Granger-causality, it is the fact that there exist such series and the number of such series is larger compared to time series without bug-fixes. In addition, we cannot conclude that files that show no Granger-causality would contradict our hypothesis, that coverage changes influence bug fixes. There can be several reasons why the Granger-causality test was not positive in these cases. A better understanding of these reasons requires further work.

There exist several time series that show Granger-causality.

Our results are also significant for SAP HANA as our comparison against non-bug-fixing-commits shows. The number of Granger-causalities on file level found for non-bug-fixing-commits is a factor 2.7 lower compared to the number of Granger-causalities found for bug-fixing commits. Therefore, we conclude that our result depends on the characterization of bug-fixing commits and therefore coverage Granger-causes bugs.

Our study complements previous studies [3], which show that testing and coverage have an impact on the improvement of code quality and the amount of code tested can be helpful in predicting which program element will require more bug-fixes. However, previous studies only highlight a correlation between coverage and bug-fixes (or mutants), whereas our findings show Granger-causality between these two variables. Further, our findings provide evidence of this relationship over a time period of 1 year instead of using only one snapshot. Thus, we have an empirical finding that supports the benefit of improving test coverage in the long run.

Our results also show that the impact of coverage is not seen immediately. The impact will become more prominent only after a certain lag. Our analysis of different lag values shows that the Granger-causality between coverage and bug-fixes is more significant after lag 3 or 4, i.e., approximately two weeks considering that coverage is created twice a week.

We further perform experiments to compare the files that show Granger-causality and are changed by bug-fixing and non bug-fixing commits. For SAP HANA, we find that the number of Granger positive files changed in bug-fixing commits is higher than Granger positive files in non bug-fixing commits. Furthermore, we observe a difference in the files that show Granger-causality changed in a bug-fixing commit against files with Granger-causality changed in a non bug-fixing commit.

On further investigation of the files which show Granger positive results and comparing them with the Granger negative files, we observe that the size of Granger positive files changes more in the time period under observation. This suggests that developers may want focus on adding test cases (and thus increasing coverage) for files whose size is likely to substantially change in the near future; such files are likely to be prone to bugs and adding test cases can prevent these bugs from being detected late. Furthermore, we observe files that show Granger-causality have a higher number of commits affecting them in the observation period.

Files with large changes have higher chances for defects.

Thus, developers may want to focus testing effort on such files, since such effort is more likely to have an impact on future bug-fixes affecting those files. All these above factors also point out that such files would be popular and indeed more testing is required, as they would have a bigger impact on the project. Granger-causality confirms that files with such characteristics indeed need to be tested more often. We do not observe significant relationships for the above metrics for Facebook React. This could possibly be due to the small number of files, i.e., 53 files that satisfy all the preconditions and 18 files that are Granger positive compared to the larger number files for SAP HANA. Nonetheless, our results for both the studied subjects show that for a substantial portion of files analyzed, a Granger-causal relationship is observed between coverage and bug fixes.

Finally, we observe that a small number of files show Granger-causality for Facebook React. In Section 4.3.3.3, React shows a weak correlation for bug fixes and this doesn't show if the assumptions of Granger-causality are violated. Once we perform the whole cycle analysis on React data, we find that the Granger-causality results are insignificant, and they do not contradict the SAP HANA results. Given that React is an open-source project, we believe that it can help other researchers replicate our study and also perform complementary studies.

#### 4.3.5.2 Threats to Validity

We discuss a list of possible threats to the validity of our work.

**Identification of Bugs:** One possible threat relates to the completeness of identified bugs, and whether the identified bugs are really bugs. We focus on bugs that are detected *late* after code is committed to a version control system. We rely on classification maintained by SAP and on keyword-based classification for Facebook React. For SAP HANA, this data is likely to be complete and correct (i.e., an identified bug is a real bug) due to the rigorous quality assurance activities done in SAP. SAP has been using this data internally for years and thus we can assume high reliability. We have also verified manually a sample of identified bugs to be valid. For Facebook data, we use Coveralls.io which is being used by several large organizations to find out uncovered parts and given that data is updated frequently.

There are two additional threats for the bug identification for Facebook React. To identify bug-fixing commits, we used a classification approach, which might not classify all bug-fixing commits correctly [117]. In addition, bug-fixing commits can contain unrelated changes the actual bug [118]. Our analysis for SAP HANA is not affected by these threats. As described in

Bug classification for React.

Section 4.3.3.1, the bug-fixing commit identification is very accurate. In addition, the development processes of SAP ensure that bug-fixing commits with such unrelated code would be rejected. Each bug-fix will be reviewed by at least one other person who would reject unrelated code changes. In addition, unrelated code changes increase the probability that a bug-fix would trigger other test failures or automated static code tests on code changes report additional problems. Therefore, developers have a strong incentive to avoid such unrelated changes for bug-fixes. The average size of a bug-fixing commit in terms of lines changed is quite low (less than 10) and also suggests that this is not an issue. To verify our assumptions, we manually checked a sample of bug-fixing commits and did not find any counterexamples with tangled code changes.

We identify bugs by bug-fixing commits. However, the time for the revelation of a fault and the time of the corresponding bug-fix may be different with an unknown gap between. For the industrial project, we expect that the length of the gap is low on average because of customer requirements. For Facebook React, we do not have additional data to estimate the length of the gap.

**Mapping between Bugs and Coverage:** Ahmed et al. [3] use the *GumTree Differencing Algorithm* [87] to determine when a program element was changed and track its history. In our analysis, we divide our observation period into multiple segments and use a moving time window approach and also explain the advantages of our approach compared to GumTree (see Section 4.3.3.2). However, it is not guaranteed that all bugs can be mapped to coverage. Note that due to the specific requirement of our approach that we map a bug to coverage on a file level, we do not need to track line differences accurately. In the case of multiple relocations of source code between different files within a single segment, our approach might be inaccurate. For SAP HANA, in 99.70% of all bugs, our approach could map the corresponding bug-fixing commits accurately to coverage.

**Flakiness of Tests:** A flaky test shows both passing and failing results for the same code. Luo et al. provide an empirical analysis of flaky tests [181]. We derive our coverage data from tests. Therefore, flaky tests affect our coverage data. A flaky test might abort early and therefore produce different coverage data for multiple runs. We also observed that generating coverage itself increases the flakiness. We expect that flakiness affects all tests in a similar way and therefore does not affect our results.

**Granger-Causality:** In this study, we use Granger-causality which is used between two variables as - a variable  $x$  is said to Granger-cause another variable  $y$  if past values of  $x$  help predict the current level of  $y$ . While Granger-causality does not represent true causality, it can help understand the relationship between coverage and the number of bug-fixes. As expressed by Geweke, Granger-causality is not identical to causation in the classical philosophical sense, but it does demonstrate the likelihood of such causation or the lack of such causation more forcefully than does simple contemporaneous correlation [99]. Stern reiterates that “Two variables may be contemporaneously correlated by chance but it is unlikely that the past values of  $x$  will be useful in predicting  $y$ , given all the past values of  $y$ , unless  $x$  does actually cause  $y$  in a philosophical sense” [239].

Related work states that Granger-causality only gives whether the relationship between variables is significant or not. It does not give the strength of causality, i.e., we do not gain information whether a specific time series  $T_1$  may have a greater or smaller influence than another time series  $T_2$  [126–128]. However, there is also a critical discussion about whether this is true [24]. We do not contribute to this dispute. Our experiment does not analyze the strength, our experiment only analysis the existence. Therefore, our results are not affected by this potential threat.

Furthermore, we investigate the relationship between coverage and bug-fixes. It is possible that other variables also correlate with bugs or coverage (e.g., we would expect that coverage correlates with test suite size). We plan to investigate such a relationship in future work.

Confounding Variables

**Generalizability:** Our work focuses on two different types of software project, however, we can not conclude that the results can be generalized. Still, to diversify our data we select both closed-source and open-source projects developed by large organizations, SAP and Facebook. Moreover, our study complements the existing body of work that has investigated the relationship between coverage and bugs. To the best of our knowledge, we are the first to analyze the Granger-causal relationship between coverage and bug-fixes in two large industrial systems. The systems that we analyze are also very large — much larger than the systems analyzed by prior work (e.g., systems analyzed by Inozemtseva and Reid are of 280K LOC on average [130]). Previous studies have also focused on one industrial system [198, 271]. Gaining access to large industrial system is a non-trivial task and we welcome future research to complement our study by replicating it on additional software systems.

#### 4.3.6 Related Work

We summarize related work on testing, coverage, and studies investigating the existence of Granger-causal relationships in software engineering.

**Coverage and Defects:** Mockus et al. study the importance of test coverage as a measure for test effectiveness on two industrial software projects and analyze the required test effort with different levels of test coverage [194]. For the two study objects, they find that an increase in coverage reduces the number of post-release defects but requires an increase in the amount of effort on testing. Ahmed et al. analyze several open-source programs to understand the correlation between coverage and bug-fixes [3]. The main hypothesis of their study is that if an element is well-tested at a given point, it should have fewer bug-fixes in the future than a poorly tested element. They find a weak yet significant correlation between statement coverage and the number of bug-fixes and program elements covered by any test case have half as many bug-fixes as those not covered. Kochhar et al. investigate two large systems to compute correlations between coverage, test suite size and test suite effectiveness using real bugs and find a moderate to strong correlation between coverage and test suite effectiveness [158].

The above studies show the need for achieving good test coverage to improve the reliability of systems. In this work, we investigate the Granger-causal relationship between test coverage and bug-fixes in a very large,

real-world software project. While the above studies analyze the correlation between coverage and test suite effectiveness and make use of mutation testing (or real bugs) for a single snapshot, we analyze Granger-causal relationship between these variables over a 1 year time frame. Furthermore, we make use of real bug-fixes instead of injecting mutants.

**Coverage and Killing Mutants:** There are several work investigating the correlation between test coverage and its effectiveness in killing mutants [3, 37, 103, 130]. Gopinath et al. experiment on hundreds of projects from GitHub [103]. They analyze manually and automatically generated test cases with mutation analysis. Manually generated test cases are collected from the projects and Randoop produces automatically generated test cases. They found that statement coverage is a good indicator of test suite effectiveness. Inozemtseva and Reid study five open-source systems and generated 31 000 test suites by randomly selecting a subset of existing JUnit tests identified using Java’s reflection API [130]. They measure statement, decision, and modified condition coverage and perform mutation testing to measure test suite effectiveness. They conclude based on the results that code coverage is not strongly correlated with test suite effectiveness. Instead of mutants, we use real bugs in a large number. Several previous studies show that mutants can be used in substitution to real bugs as they are similar to real faults [8, 147], while other studies show that mutants may not be representative of real bugs [104, 200]. Furthermore, there are limitations and problems like the ease of detection [8] and issues related to subsuming and subsumed mutants [210]. Considering the differing views for the relationship between mutants and real faults, we use real faults as reported in the issue tracking system. This can help prevent any threats that might affect a study due to the usage of mutants.

**Other Studies:** Kochhar et al. investigate over 300 large open-source projects from GitHub to analyze the correlations between coverage and metrics such as lines of code, cyclomatic complexity and the number of developers [159]. Tengeri et al. propose an approach for test suite assessment and improvement based on code coverage and use it for purposes such as removal, refactoring and extension of test cases [243]. In this study, we analyze a very large software project which is developed closed-source by SAP to understand Granger-causality between coverage and bug-fixes.

**Studies on Causal Relationship:** Couto et al. study the Granger-causality between software metrics and software defects [54–56]. They apply the Granger-causality test (see Section 4.3.2.2) as a statistical hypothesis test to investigate whether past variations in source code metrics can forecast changes in defects. They analyze four Java projects and collect various types of metrics such as CK metrics, LOC, number of methods (NOM). They identify bugs by linking issue tracking system entries to commit logs by finding references of commit hashes in issues and vice versa. They first check for stationary time series and then apply the Granger-causality test to measure the impact of metrics on the number of bugs. They further compute threshold values of different metrics and use that as an input to a defect prediction model to provide recommendations to developers on files that are likely to have bugs in the future. They were able to find causes for 64% to 93% of the defects. Canfora et al. analyze four open-source Java

and C systems and apply the Granger-causality test to understand whether a change in a software artifact is related to changes occurred in some other artifacts [39]. They find that a hybrid recommender using combinations of association rules and Granger-causality can achieve higher recall than the two techniques used individually. Different from above studies, we examine Granger-causality between coverage and the number of bug-fixes.

**Studies on Industrial Systems:** Studies on industrial systems are important for practitioners and researchers to identify new problems and gain new insights. To our knowledge, there does not exist a previous study on Granger-causality for a very large industrial system. Memon et al. study continuous integration (CI) at Google [191]. They focus on analysis and improvements for the CI system at Google, but also provide results about relationships among frequency of changes, file types, number of authors and test results. For example, their data shows that a single file with many changes by different authors has an almost 100 % probability to cause a failure. Zimmerman et al. investigate how dependencies correlate with defects and use this to predict defects for binaries in Windows Server 2003 [271]. Our study complements these studies by analyzing Granger-causal relationship between coverage and bug-fixes in a large industrial system.

#### 4.3.7 Conclusion and Future Work

Testing, being an integral part of software development, is a continuous process that helps improve the quality of the underlying system. Code coverage metric is often used as a proxy for the amount of testing. Past studies have analyzed coverage and test suite effectiveness, however, those studies are mostly limited to analyzing correlation, using mutants and collecting a single snapshot. Different from previous studies, we investigate the existence of Granger-causal relationship between change in coverage and the number of bug-fixes by collecting longitudinal data over 1 year time period for SAP HANA and Facebook React. For every source file, we compute coverage for regression test runs made in the observation period, and the number of bug fixes made between regression test runs. After performing several preprocessing steps, we apply the Granger-causality test and further examine the characteristics of files that show Granger-causal relationship between change in coverage and the number of bug fixes.

Our empirical study leads to the following findings:

1. 29.45 % of the files (1 893/6 428) for SAP HANA and 33.96 % of the files (18/53) for Facebook React exhibit Granger-causality between coverage ratio time series and number of bug-fixes time series.
2. The effect of coverage changes on bug-fixes does not show immediately. The impact becomes significant after a certain delay. For lag values of 3 to 4, the number of files showing Granger-causality is the highest compared to all other lag values from 1 to 10.
3. For SAP HANA, the Granger-positive files are statistically significantly different from the Granger negative files in several aspects. These files have a higher churn during the observation period and more commits affect them. We obtain non-conclusive results for Facebook React likely due to the small number of data points available for analysis.

Future work could investigate a larger dataset of projects to reduce threats to external validity. Furthermore, it could be interesting to investigate differences between Granger positive and negative files in terms of additional factors such as the size of project, development environment, team composition, company culture, programming languages, or project domain. It is unclear to us whether such factors could be confounding variables.

In our work, we utilized historical information about bugs and bug-fixes. As explained in our introduction, it is unclear to us whether mutants could be a suitable replacement for such bug data. An analysis of Granger-causality for mutants and coverage could provide further insights into the question of how suitable mutants are as a replacement for bugs.

#### 4.4 *Combinatorial Testing*

Test suites in complex software projects might grow over time to considerable sizes. This results in incurring high maintenance effort and prolonged execution times. Maintaining their quality and efficiency requires pruning of redundancies while increasing, or at least retaining their coverage levels.

We propose a lightweight method to tackle these problems with a focus on condition/decision coverage (C/D coverage). First, we describe a method to reduce the size of unit test suites while preserving their C/D coverage degree. We then introduce an approach which combines combinatorial testing and input space modeling to further increase the degree of the C/D coverage.

Combine combinatorial testing with equivalence partitions.

Our semi-automated method works even in the absence of models or documentation and produces a low number of new test cases that require the creation of new test oracles. We also do not use symbolic execution techniques. These properties make our approach practically applicable for industrial projects and simpler to implement.

We evaluate our approach on selected examples from SAP HANA. We demonstrate that it is possible to generate from integration tests new suites of unit tests with high C/D-coverage but with only a few test cases. At the same time, the human effort of creating such suites is moderate.

##### 4.4.1 *Introduction*

Researchers have proposed a wide range of techniques to automatically create tests covering untested code. These techniques can be summarized as coverage-adaptive test creation. One family of techniques for coverage-adaptive test creation regards application code as a black-box and, after test creation, verifies whether test coverage within the black-box has increased or not. Another option is to analyze the source code as a white-box, and to create suitable tests based on knowledge of the code that reach uncovered parts of the source code. A popular technique for the latter group utilizes constraint solvers or SAT solvers to determine the inputs that trigger the execution of certain parts of the code. However, such approaches have practical limitations due to highly structured inputs, external libraries, and large complex program structures. These factors pose considerable challenges for obtaining solutions in an effective way — despite the NP-completeness of the underlying decision or constraint satisfaction problems.

Creating new tests is a complex task.



We propose to use combinatorial testing as a technique for the automatic generation of minimal amount of new tests needed to increase the condition/decision coverage of an existing test suite. Our approach first trims the number of test cases in the suite while maintaining the degree of condition/decision coverage. It then collects the input values used in the remaining tests and uses them as input levels in the combinatorial test generation. In other words, we do not assume the availability of a model or even documentation while applying combinatorial testing.

We further reduce the covering arrays created by combinatorial testing techniques to a minimal test set that enhances the condition/decision coverage of an existing test suite. Based on the assumptions that existing test inputs have some purpose and new combinations of them consist of partly meaningful input, we expect that these newly generated tests have an advantage compared to random tests. In particular, they can cover untested parts of the source code, because the original tests already contain some domain knowledge about the internal behavior of the system under test.

Our technique has fewer requirements than approaches based on symbolic execution. We only require test inputs and coverage data. In comparison, symbolic execution requires a complex analysis of the SAT problem and a precise syntactic understanding of the source code.

Simpler than symbolic execution

We show the results of a preliminary evaluation of our approach on two examples from SAP HANA. The evaluation indicates that our technique can improve existing test suites in terms of condition/decision coverage. However, it is not able to create full coverage in all scenarios. For such cases, we revert to the manual design of input levels with additional insights from previous steps of our technique.

Our contributions are the following ones:

- A method for improving the level of condition/decision coverage of an existing test suite via combinatorial testing while maintaining small suite size and requiring only a few test oracle queries that is independent of the source code of the system under test.
- A preliminary evaluation of our approach on functions from SAP HANA.

#### 4.4.2 Methodology

In this section, we first describe a method for reducing the size of unit test suites while maintaining their degree of condition/decision coverage (Section 4.4.2.2). In Section 4.4.2.3 we then show how the level of the condition/decision coverage can be increased by exploiting combinatorial testing and input space modeling. Fig. 4.12 gives an overview of the complete approach. The major processing steps are:

1. Reducing the original test suite  $B$  to  $B_r$  by solving with our implementation from Section 3.3.4 a set cover/hitting set problem (Section 4.4.2.2).
2. Generating new tests for  $B_r$  by combinatorial testing using projections of input tuples (Section 4.4.2.4).
3. Optional: Generating combinatorial tests with the support of input space modeling if required (Section 4.4.2.5).
4. Reducing the test suite again (Section 4.4.2.6).

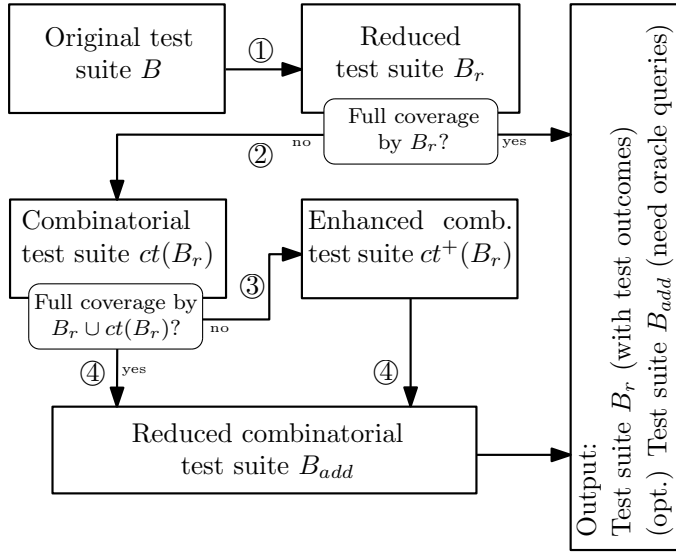


Figure 4.12: Overview of the processing flow of the complete approach.

4.4.2.1 Definitions and Conventions

In the following, we assume that we start with a given suite of unit tests for a target function  $f$ . By analyzing (or instrumenting) these tests we can collect for each execution of  $f$  the values of all arguments passed to  $f$ , and the return value of  $f$ . We call such a record an *input/output pair* and call the group of input values an *input tuple*.

Input/output pair

We analyze these input/output pairs and retain only those which are unique in terms of the input tuples (i.e., after this step each input tuple for  $f$  occurs only once). Since the considered functions are deterministic, this is equivalent to computing unique elements over the input/output pairs. We write  $B(f)$  or just  $B$  for the resulting set of pairs.

Note that each element of  $B$  completely characterizes a unit test for  $f$  (including the expected test result). In the following, we use the notions input/output pair and unit test interchangeably for the elements of  $B$ .

For the terms related to coverage, the definition we use follows Section 3.1.5. However, we reiterate it and provide additional examples to highlight specific attributes of the definitions. A Boolean expression in any branch or a loop condition of the considered code is called a *decision*. A *condition* is a leaf-level Boolean expression that cannot be broken down into simpler Boolean expressions. Thus, a decision is either a condition, or it consists of several conditions and Boolean operators. For example, a decision like  $y < y1 \ || \ y == y1$  gives rise to two conditions:  $y < y1$  and  $y == y1$ . Fig. 4.14 and Table 4.10 illustrate this further. For example, decision  $D2$  (line 5) generates conditions  $C2$ ,  $C3$ , and  $C4$ .

Decision  
Condition

We call a combination of a decision or a condition  $X$  with an outcome  $y$  a *expression-outcome pair* and write  $X = y$  for it. When a test executes a decision (or condition)  $X$  and the outcome of the evaluation is  $y$ , we say that this test case *covers* the expression-outcome pair  $X = y$ . In Fig. 4.13, test  $U_1$  covers  $D1 = f$ ,  $D2 = f$ , and four other expression-outcome pairs.

Expression-outcome pair

For a set  $S$  of tests and an expression-outcome pair  $X = y$ , let  $cov_{X=y}(S)$  be a set of those tests in  $S$  which cover  $X = y$ . For example, for a condition  $C1$  with outcome *true* the collection  $cov_{C1=t}(S)$  contains all tests from  $S$

Cover.	Expr.	$U_1$	$U_2$	$U_3$	$U_4$	$U_5$	$U_6$	...	$U_{171}$
✓	$D1, f$	1	1			1		...	1
✓	$D1, t$			1	1		1	...	
✓	$D2, f$	1		-	-	1	-	...	
✓	$D2, t$		1	-	-		-	...	1
✓	$D3, f$	-	-			-	1	...	-
✓	$D3, t$	-	-	1	1	-		...	-
✓	$C1, f$	1	1			1		...	1
✓	$C1, t$			1	1		1	...	
✗	$C2, f$			-	-		-	...	
✓	$C2, t$	1	1	-	-	1	-	...	1
✗	$C3, f$			-	-		-	...	
✓	$C3, t$	1	1	-	-	1	-	...	1
✓	$C4, f$	1		-	-	1	-	...	
✓	$C4, t$		1	-	-		-	...	1
✓	$C5, f$	-	-			-	1	...	-
✓	$C5, t$	-	-	1	1	-		...	-

Figure 4.13: An illustration of the expression-outcome pairs, their coverage, and a minimum hitting set  $B_r$  for the function  $f1$  in Fig. 4.14. A row  $X, y$  corresponds to an expression-outcome pair  $X = y$ , and a column  $U_i$  to a unit test from  $B$ . An entry 1 for row  $X, y$  and column  $U_i$  means that a condition or decision  $X$  is executed and evaluated to  $y$  under test  $U_i$ . Column “Cover.” indicates whether test suite covers (✓) the expression-outcome pair  $X = y$ , or not (✗). The set  $B_r = \{U_1, U_2, U_4, U_6\}$  is a minimal hitting set.

which execute  $C1$  and evaluate it to *true*. In Fig. 4.13, the set  $cov_{D1=f}(B)$  are all the unit tests  $U_i$  (columns) which have a 1 in their first row.

A test suite achieves a full *condition/decision coverage* if (while executing all test cases in the suite) “*every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, and every decision in the program has taken all possible outcomes at least once*” [45, 46, 145]. See also Section 3.1.5 for further discussion of various coverage variants.

Condition/decision coverage

#### 4.4.2.2 Test Suite Reduction with Unchanged C/D-Coverage

The purpose of the following approach is to identify a minimal (or at least a small) subset of unit tests from  $B$  which (cumulatively) have the same condition/decision coverage (C/D coverage) as  $B$ .

Our approach works in the following steps:

1. We first identify all decisions  $D_1, \dots, D_k$  in the target function  $f$  and split each of these decisions into conditions  $C_1, \dots, C_l$ . Currently, this step is performed manually but can be automated by using a parser.
2. For each condition and each decision, we instrument the source code to collect data whether this expression has been evaluated and if yes, what was the Boolean result. This step is only necessary to compensate that our dynamic code analysis tool *DynamoRio* is currently not configured to collect condition and decision coverage data.
3. For each input/output pair in  $B$  we generate a corresponding unit test and subsequently execute it. At each execution, we collect data for all conditions and decisions described in the previous step.

Obtain decisions and conditions.

Obtain coverage.

4. For each expression-outcome pair  $X = y$  in the target function  $f$  we identify from data generated in the previous step all unit tests in  $B$  which cover  $X = y$ . In other words, we compute the sets  $cov_{X=y}(B)$  over all expression-outcome pairs in  $f$ . Obtain mapping.

The outcome of this step is a collection of  $k \leq 2p$  sets of tests, where  $p$  is the total number of identified decisions and conditions in a target function  $f$ . If any of the computed sets is empty (i.e.  $k < 2p$ ), the corresponding expression-outcome pair is not covered. Fig. 4.13 illustrates that for Example 1 the original unit tests have already covered  $k = 14$  out of  $2p = 16$  expression-outcome pairs, and thus only two sets  $cov_{C2=f}(B)$  and  $cov_{C3=f}(B)$  are empty.

5. Finally, for the test suite reduction we compute an exact or approximate *minimal hitting set*  $B_r$  [148]. Here  $B$  is the universe and the subsets to be “hit” are the  $S_1, \dots, S_k$  (non-empty) sets  $cov_{X=y}(B)$  computed in the previous step. By the definition of the minimal hitting set problem,  $B_r$  has at least one element in each subset  $S_i$  (this element “hits”  $S_i$ ), and has the smallest size among all subsets of  $B$ . In other words, a test suite induced by  $B_r$  is guaranteed to have the same C/D-coverage as the full test suite  $B$ . Typically,  $B_r$  has substantially less elements than  $B$ . Reduce tests.
- In Fig. 4.13, set  $B_r = \{U_1, U_2, U_4, U_6\}$  is a minimal hitting set for the universe  $B$  and the sets to be hit being all non-empty sets  $cov_{X=y}(B)$  (i.e., all except for  $cov_{C2=f}(B)$  and  $cov_{C3=f}(B)$ ).

To compute the minimal hitting set we convert the problem to an equivalent set cover problem and use the approach presented in Section 3.3.4.

The resulting test suite  $B_r$  is possibly not achieving the full C/D-coverage, if  $B$  was not fully covering. To increase the degree of coverage we apply combinatorial testing enhanced by input space modeling.

#### 4.4.2.3 Improving C/D-Coverage via Combinatorial Testing

For a set  $S$  of input/output pairs, let  $P_i(S)$  denote the set of all unique values encountered as the  $i$ -th component of any input tuple from  $S$ . In other words, such a *projection on  $i$ -th argument* is the set of all different values used as  $i$ -th input argument for a target function  $f$  while executing test suite corresponding to  $S$ . For example, if  $f$  has 3 arguments  $x, y, z$  and  $S$  contains the input/output pairs  $((1, 2, 3), 6)$ ,  $((5, 2, 1), 8)$ , and  $((10, 3, 7), 20)$ , the projection  $P_1(S)$  is  $\{1, 5, 10\}$  and the projection  $P_2(S)$  is  $\{2, 3\}$ .

By examining the original  $B$  or the reduced test suites  $B_r$  we observed that expression-outcome pairs  $X = y$  are more frequently not covered if  $X$  is a condition and not a decision (the higher decision coverage can be attributed to the high line coverage in our samples). Among the uncovered expression-outcome pairs with conditions we identified two types:

1. A “complex” condition like `b <= (to - from + 1)`, where two or more input arguments are involved directly or indirectly (i.e., if one of the variables is derived from arguments but not an argument itself). In such a case, there is no coverage because in  $B_r$  there is no suitable *combination* of input values for involved arguments. Our approach (Section 4.4.2.4) attempts to solve this issue by using combinatorial testing to create more

combinations of values for these arguments. For each argument, we use as values only the projections of  $B_r$  (i.e., original values for this argument). If this is not sufficient, we proceed as in Section 4.4.2.5.

2. A “simple” condition like `variable == val` (or e.g., `variable < val`), where `variable` is a function argument or a variable derived from a single function argument. In such a case the condition is not fully covered because there is no appropriate value in the projection of  $B_r$  on the argument relevant for `variable`. In this case, combinatorial testing based on projections of  $B_r$  cannot help and we need to introduce new values for the relevant argument (Section 4.4.2.5).

#### 4.4.2.4 Generating Combinatorial Tests From Projections

Our scenario does not assume knowledge about a model for a target function  $f$ , which makes the traditional model-based input space modeling not applicable. Instead, we derive levels for each argument of  $f$  (i.e., concrete values for this argument used in the combinatorial test suite generation) as follows: levels for  $i$ -th argument are exactly all different values used for this argument in the test set  $S$ . In other words, the levels for the  $i$ -th argument of  $f$  are the projections  $p_i(S)$  of  $S$  on  $i$ -th argument.

We then check whether there are any uncovered complex conditions and for each such a condition we determine the number  $m$  of involved input arguments. For the subsequent test generation, we then set the interaction level  $t$  as the maximum of all uncovered complex conditions. To obtain a combinatorial test suite, we generate a covering array with a tool like *ACTS* [262] with a pre-computed interaction level  $t$ .

Note that this method works with the original set  $B$  of all logged unit tests and the reduced suite  $B_r$  maintaining the original C/D-coverage level. Since  $B$  is likely to have larger projections per argument, we only generate a combinatorial test suite over  $B_r$ . Given a fixed interaction level  $t$ , we write  $ct_t(B_r)$  or just  $ct(B_r)$  for the set of generated unit tests in this step.

In the subsequent step, we check the C/D coverage of the generated combinatorial suite. If some conditions or decisions are still not covered, we additionally execute the steps described in the next subsection.

#### 4.4.2.5 Input Modeling for Combinatorial Tests

To improve the coverage of uncovered decisions/conditions, it can be necessary to perform manual input space modeling prior to generating a combinatorial test suite. We collect the identical levels  $p_i(S)$  as in Section 4.4.2.4. Then, for each uncovered simple condition (e.g., `variable == val`) we add a level which evaluates this condition to the missing *true/false* outcome.

For complex conditions, we consider the projections for all involved input arguments, and then add level(s) such that at least one combination (of original and new levels) evaluates the condition to a missing *true/false* value. We also set the minimum interaction level  $t$  to the number of involved input arguments. A similar approach is taken for the decisions.

Overall, there is no guarantee that suitable levels will be found and this process is highly manual. In future work, we plan to use constraint satisfaction methods to automatize this part.

Generate new tests from new combinations of existing values.

Enhance input values manually or with symbolic execution.

After the levels for all arguments are fixed, we generate a combinatorial test suite (with pre-specified interaction level  $t$ ) similarly as in Section 4.4.2.4. We designate it by  $ct_t^+(B_r)$ , or just by  $ct^+(B_r)$ .

#### 4.4.2.6 Reducing Combinatorial Test Suites

A combinatorial test suite  $T$  (either  $ct(B_r)$  by method in Section 4.4.2.4 or  $ct^+(B_r)$  by method from Section 4.4.2.5) typically has more test cases than the original set  $B_r$ . Moreover, most of the combinatorial unit tests have input combinations not in  $B$ , and thus require queries to a test oracle.

Therefore, we apply again the test suite reduction technique described in Section 4.4.2.2, Step 5. However, the universe is now the set of input tuples corresponding to the tests in the test suite  $T$  and the sets  $S_1, \dots, S_k$  to be “hit” correspond *only* to the conditions and decisions not covered (not “hit”) by the reduced test suite  $B_r$ .

For example,  $f$  has the three decisions  $D_1, \dots, D_3$  and the five conditions  $C_1, \dots, C_5$ .  $B_r$  (or, equivalently  $B$ ) does cover neither of the two cases  $C2 = false$  and  $C3 = false$ . We would perform here the test suite reduction only with  $cov_{C2,false}(T)$  and  $cov_{C3,false}(T)$  (i.e., computed over  $T$ ) as sets to be hit. This ensures that the resulting reduced test suite ( $B_{add}$ , say) contains only unit tests which handle condition/decision and outcome combinations which are *not yet covered* by  $B_r$ .

The overall output of our approach is a test suite with all the tests from  $B_r$  and additional tests from the just reduced suite  $B_{add}$ . The earlier tests cover the same expression-outcome cases as  $B$  while the latter attempt to cover additional cases. Note that we need to query the test oracle for all new tests in  $B_{add}$ , but not those from  $B_r$ .

Test reduction repeated.

#### 4.4.3 Experiments and Evaluation

We evaluate our technique on example functions of SAP HANA (see Section 2.2). Due to the large size of SAP HANA, we have a large set of tests we can use to extract input values for our approach (see Chapter 6).

##### 4.4.3.1 Improvements of C/D-coverage

We evaluate our approach on several examples from the SAP HANA test suite, and discuss here two of them. For reasons of confidentiality, we partially obfuscate the source code and the variable names.

**Example 1:** The first example is a function  $f1$  shown in Fig. 4.14. We identify three decisions and five conditions as shown in Table 4.10.

```

1 size_t f1(size_t from, size_t to, size_t a, size_t b){
2   if (from == 1) {
3     return (b < to) ? b : a;
4   }
5   if (b > 0 && b <= (to - from + 1) && (a < from + b)) {
6     return from + b;
7   }
8   return a;
9 }

```

Figure 4.14: Source code of Example 1, partially obfuscated.

C/D Name	Expression	C/D Name	Expression
<i>D1</i>	(Line 2)	<i>C2</i>	$b > 0$
<i>D2</i>	(Line 5)	<i>C3</i>	$b \leq (\text{to} - \text{from} + 1)$
<i>D3</i>	(Line 3)	<i>C4</i>	$a < \text{from} + b$
<i>C1</i>	$\text{from} == 1$	<i>C5</i>	$b < \text{to}$

Table 4.10: Conditions and decisions in Example 1; “(Line  $k$ )” indicates that the decision can be found in line  $k$  of Fig. 4.14.

Table 4.11 shows intermediate results after five essential phases of the algorithm. Phase *a* refers to analyzing the original tests and the corresponding column shows statistics for the test suite  $B$ . There are 171 unit tests but two expression-outcome cases are not covered, namely  $C2 = false$  and  $C3 = false$  (see e.g. row  $C2$ , where only 19 from 171 unit tests execute this condition, and all outcomes are *true*).

Phase	a	b	c	d	e
Name	$B$	$B_r$	$ct(B_r) \cup B_r$	$ct^+(B_r) \cup B_r$	$B_{add}$
#tests	171	4	36	38	2
#oracle	0	0	32	34	2
#missing	2	2	1	0	0
<i>D1</i>	152/19	2/2	28/8	5/33	-
<i>D2</i>	6/13	1/1	4/4	10/23	-
<i>D3</i>	131/21	1/1	22/6	4/1	-
<i>C1</i>	152/19	2/2	28/8	5/33	-
<i>C2</i>	19/0	2/0	8/0	27/4	1/1
<i>C3</i>	19/0	2/0	5/3	17/12	0/1
<i>C4</i>	6/13	1/1	4/1	10/7	-
<i>C5</i>	131/21	1/1	22/6	4/1	-

Table 4.11: Intermediate results of the approach for Example 1 for the phases *a* to *e* described in text. In row *Name*, #tests states the number of test cases in the suite, #oracle gives the number of tests which need an oracle query, #missing states the number of uncovered expression-outcome pairs. Each of the remaining rows corresponds to a decision or a condition  $X$ , and for each phase the entry  $p/q$  says that the suite had  $p$  tests which covered the expression-outcome pair  $X = true$  (i.e.,  $|cov_{X=true}| = p$ ), and there were  $q$  tests which covered the expression-outcome pair  $X = false$  (i.e.,  $|cov_{X=false}| = q$ ).

Phase *b* refers to the first reduction (Section 4.4.2.2) and shows numbers for  $B_r$ . The four tests in this suite have the same C/D-coverage as  $B$ . Yet, both  $C2 = false$  and  $C3 = false$  are still not covered. Phase *c* is the combinatorial test generation from projections (see Section 4.4.2.4).

To ensure that this step does not reduce the total coverage, we add the tests from  $B_r$  to the union  $ct(B_r) \cup B_r$ . This is necessary because the generated suite of the combinatorial tests might not contain the tests from which projection is taken. In this step, combinatorial testing indeed manages to increase the C/D-coverage. Out of 32 tests found by combinatorial testing for 100% 2-way coverage [262], 3 tests cover  $C3 = false$ .

Phase *d* refers to input modeling for combinatorial tests (Section 4.4.2.5). We provide statistics for  $ct^+(B_r) \cup B_r$ . By adding a new level 0 for argument  $b$  we can cover  $C2 = false$  (since condition  $C2$  is “ $b > 0$ ”). After this phase, all conditions and decisions are covered, but there are too many (combinatorial) tests - 34 new tests.

In phase *e* we reduce these new CT-generated tests (i.e.,  $ct^+(B_r)$ ) as described in Section 4.4.2.6, but only for the following two sets to be hit:  $cov_{C2=false}(B_r)$  and  $cov_{C3=false}(B_r)$ . The result is a test suite  $B_{add}$  with two test cases, which covers both  $C2 = false$  and  $C3 = false$ .

Note that the final test suite contains 6 test cases. 4 test cases (in  $B_r$ ) have already test results and 2 (in  $B_{add}$ ) require queries to test oracle.

We conclude that for example 1, the derived unit tests already produced a high C/D-coverage with 14 of 16 expression-outcome pairs covered. The reduced test suite features the same C/D-coverage but has only 4 (down from 171) unit tests. “Basic” combinatorial testing (Section 4.4.2.4) covers one more expression-outcome pairs. Further input modeling (Section 4.4.2.5) allows covering the last open case. We conclude that both coverage enhancement via “basic” combinatorial testing and in combination with input modeling are useful in this case. They offer a trade-off between the degree of human involvement (for input modeling) and level of the C/D-coverage.

**Example 2:** The second example is a function *f2* that essentially consists of a single decision using short circuit evaluation as shown in Fig. 4.15. We identify a single decision with nine conditions as shown in Table 4.12.

```

1 bool f2(int y1, int m1, int d1, int h1, int mi1,
2         int y2, int m2, int d2, int h2, int mi2) {
3     return y1 < y2 ||
4         (y1 == y2 && (m1 < m2 ||
5         (m1 == m2 && (d1 < d2 ||
6         (d1 == d2 && (h1 < h2 ||
7         (h1 == h2 && mi1 < mi2))))));
8 }

```

Combinatorial testing improves C/D coverage.

Figure 4.15: Source code of Example 2, partially obfuscated.

C/D Name	Expression	C/D Name	Expression
<i>D1</i>	(Lines 3-7)	<i>C5</i>	<i>d1</i> < <i>d2</i>
<i>C1</i>	<i>y1</i> < <i>y2</i>	<i>C6</i>	<i>d1</i> == <i>d2</i>
<i>C2</i>	<i>y1</i> == <i>y2</i>	<i>C7</i>	<i>h1</i> < <i>h2</i>
<i>C3</i>	<i>m1</i> < <i>m2</i>	<i>C8</i>	<i>h1</i> == <i>h2</i>
<i>C4</i>	<i>m1</i> == <i>m2</i>	<i>C9</i>	<i>mi1</i> < <i>mi2</i>

Table 4.12: Conditions and decisions in Example 2. Decision *D1* can be found in lines 4 to 8 of Fig. 4.15.

Table 4.13 shows intermediate results for *f2* (same format as Table 4.11). The extracted unit tests (i.e., *B*) cover all expression-outcome pairs except for *C9* = *true*. Unfortunately, none of the two variants of coverage enhancement was successful in this case. However, there is an input tuple (unit test) that covers this case, which we derive manually (see below).

First, we discuss the input modeling step (Section 4.4.2.5). With the levels for *mi1* and *mi2* derived from *B<sub>r</sub>* it is not possible to evaluate condition *C9* (*mi1* < *mi2*) to true, as all levels for *mi1* are larger 0 and *mi2* is always 0.

Therefore, we add 10 as a new level for *mi2* to the input set for combinatorial testing. We calculate new result sets for 100% 2-way coverage and 100% 4-way coverage. Both result sets did not cover *C9* = *true*. We expect that (at latest) a test suite with a full 10-way coverage should contain a suitable input, but the tool which we use does not allow to use an interaction level *t* larger than 6 (and could output at most 10 000 test cases).

Instead, we manually derive an input tuple which satisfies *C9* = *true* by modifying the input tuple in *B<sub>r</sub>* which covers *C9* = *false* (namely 2008, 10, 5, 2, 5, 2008, 10, 5, 2, 0): we replace the last argument value 0 by 10 (any value larger 5 would work). Obviously, such cases can be solved by a constraint solver by seeding it with the additional information we already have from execution. We plan to investigate this option in our future work.

Manual enhancement required.



Phase	a	b	c	d	e
Name	$B$	$B_r$	$ct(B_r) \cup B_r$	$ct^+(B_r) \cup B_r$	$B_{add}$
#tests	256	9	56	57	0
#oracle	0	0	47	48	0
#missing	1	1	1	1	1
$D1$	113/143	5/5	34/22	34/22	-
$C1$	12/244	1/8	28/28	28/29	-
$C2$	194/50	7/1	10/18	11/18	-
$C3$	51/143	1/6	3/7	4/7	-
$C4$	96/47	5/1	6/1	6/1	-
$C5$	27/69	1/4	2/4	2/4	-
$C6$	46/23	3/1	3/1	3/1	-
$C7$	23/23	1/2	1/2	1/2	-
$C8$	17/6	1/1	1/1	1/1	-
$C9$	<b>0/17</b>	<b>0/1</b>	<b>0/1</b>	<b>0/1</b>	-

Table 4.13: Intermediate results of the approach for Example 2 for the phases a to e described in text. The values have the same meaning as in Table 4.11.

#### 4.4.4 Related Work

Our work is broadly related to test amplification, symbolic execution, test coverage, and combinatorial testing. Test amplification describes a set of approaches targeting improving test quality [60]. Related terms are test augmentation and test enhancement.

Our approach creates new tests to maximize condition/decision coverage and can be interpreted as an alternative to symbolic execution. The latter technique is one of the most frequently mentioned techniques for test creation since 2000 [208]. However, despite the excitement and amount of novel research, symbolic execution has practical limitations due to highly structured inputs, external libraries, and large complex program structures [208]. Our approach has fewer limitations for practical applications and utilizes combinatorial testing as an alternative to symbolic execution techniques to tackle complex program structures.

Work of Bloem et al. [26] is probably closest related to our study. In fact, they apply similar algorithmic steps as our approach. They utilize a set of existing test cases to iteratively create new test cases until a termination criterion is met. They generate a new test case with a backtracking constraint solver that iterates over all possible execution paths until it reaches a desired branch of the source code. The evaluation shows that this approach can create up to 100% branch coverage. However, their approach is applicable only under certain preconditions. For example, changes to source code should be possible and long execution times acceptable.

Coverage criteria in general and C/D coverage used in our work can be automatically measured and compared for different approaches leading to a plethora of studies [115, 211]. However, Staats et al. point out that coverage criteria such as MC/DC coverage can be ineffective for determining test suite adequacy [236]. Inozemtseva et al. conclude that coverage is not strongly correlated to test suite effectiveness when test suite size is controlled [130]. In contrast, our work on the correlation between coverage and bugs in Section 4.2 shows that coverage correlates to future bugs [17]. Therefore, it

Symbolic execution.

Backtracking constraint solver

Usefulness of coverage.

may be beneficial to create tests for uncovered and therefore untested parts of the source code – at least for our system under test.

Several research work analyze combinations between combinatorial testing and coverage. Choi et al. study the code coverage (line, branch) effectiveness of combinatorial  $t$ -way testing with small  $t$  for grep, make, and flex in different versions [47]. They found that  $t$ -way testing creates already an efficient coverage compared to exhaustive testing for  $t < 5$ . Czerwonka investigates the effects of combinatorial testing on coverage for several utility programs in Microsoft Windows [59]. Czerwonka concludes that full  $t$ -way coverage can lead to test suites with same  $t$ -way coverage but different (line, statement) code coverage. We can confirm this observation from our experience with the examples. However, we expect this effect to decrease when the interaction level  $t$  becomes comparable to the number of arguments.

Combinatorial testing and coverage.

#### 4.4.5 Summary Combinatorial Testing

We propose an approach to reduce the size of test suites and enhance them with new tests created by combinatorial testing techniques. Our preliminary evaluation shows that our method can improve the condition/decision coverage for two examples while reducing the size of the test suite. Additionally, only a few new tests require test oracle queries. Our approach has lower technical requirements compared to symbolic or concolic execution. However, our approach has also limitations. For example, in some scenarios, it could not generate additional test cases to achieve full C/D coverage.

Understanding the robustness and applicability of the proposed technique requires further analysis such as an evaluation on multiple projects and a larger set of functions. As discussed in Section 4.4.3, we also see potential benefits of approaches that combine measuring code coverage, combinatorial testing, and symbolic execution.

### 4.5 Conclusions

Our work adds additional data points to the question of the relationship between coverage and defects. We have shown for a large industrial project that the amount of defects within the source code areas covered by tests is less than we would expect if they would follow a uniform distribution. Additionally, we have shown for a large industrial project that coverage time series and defect time series show Granger-causality. Finally, we have shown how we can combine coverage and combinatorial testing to increase the test coverage of a program with less effort than other test creation techniques.

In conclusion, we believe that measuring and using code coverage can be beneficial for various activities in software engineering. Therefore, we partially answered the starting question of this chapter whether coverage data provides any additional information on faults.

However, even with considering our results, we do not claim a final answer to this question. We strongly believe that further studies are required to increase construct validity and generalizability of results due to a large amount of possible confounding variables.

# 5 | Analysis of Approaches for Test Cost Reduction

Test cost reduction is important for large software projects as discussed in Chapter 2. Consequently, there exists a wide range of research work on this subject. However, previous work typically targets small- to medium-sized projects. Approaches for test cost reduction that work for such smaller projects may not be effective for large projects. Even more, issues that are not visible or ignored for smaller projects can be amplified by the size of large projects to considerable problems that affect test costs significantly as discussed in Section 2.2.3. In this chapter, we present and analyze several approaches to tackle these issues. The main contributions are:

- An approach for test cost reduction that tackles the superlinear increase of test costs over time.
- An analysis of a test case prioritization approach based on coverage and test execution times for a large project.
- An evaluation of redundancy removal on coverage data for a large project.
- Multiple approaches for shared coverage detection and test core identification and their evaluation.
- A discussion of unsolved problems such as random coverage and flakiness.

## 5.1 Background

As discussed in Section 2.1.4, there is a wide range of research for test cost reduction. However, we faced multiple challenges in applying them for our study subject SAP HANA. We briefly discuss challenges for the categories introduced in Section 2.1.4: test case prioritization (TCP), regression test selection (RTS)/test case selection(TCS), and test suite reduction (TSR).

Approaches that reduce test cost by identifying a fraction of tests that can be removed (category TSR) or skipped (category TCS) do not scale for large projects. Even if 10% of all tests can be skipped for a test run, the remaining 90 000 might still be too large. Even more, in the case the amount of tests continues to grow over time, such a constant reduction cannot provide sufficient reductions over time. Therefore, it is necessary to find techniques that tackle the growth of tests over time.

Approaches that prioritize tests (category TCP) provide little usefulness in practice for large projects. While it sounds tempting to prioritize tests in such a way that tests finding defects execute first and test runs will abort after finding the first defect, the practical savings are low. First, the savings only apply in negative cases, i.e., if a change introduces a defect. In a positive case, all test cases are executed (in a specific order). Second, flaky tests

invalidate the underlying assumptions that a test failure indicates a defect. In fact, in large projects, each test run can contain several hundred tests that failed due to flakiness. Aborting a test run after the first failure would result in a state of staleness. Therefore, it is required to design approaches that are not affected by flakiness or find ways to reduce flakiness.

As we conducted our research on a large software project, we encountered several additional issues for existing approaches:

1. Approaches that expect frequent coverage information cannot be applied in practice. Generating coverage for a large number of tests is time-consuming and costly in terms of the required hardware.
2. Examining a source code repository for historical data can be a complex task due to a rather complex development process.
3. The *choice of algorithms for TCP/TCS* has a large impact on the results. Choice of algorithms for TCP/TCS
4. *Shared coverage* complicates the analysis of coverage data. Shared coverage are parts of the software executed by all or nearly all tests. Shared coverage
5. The coverage size is rarely considered in the design of algorithms working with coverage data. With coverage data size of 1 GB to 100 GB, algorithms that require an execution time that depends superlinearly or even exponentially on the size of coverage are not practical anymore.
6. Random coverage does not allow precise analysis.
7. Flakiness affects any approach that assumes the correctness of test results.

We elaborate on several of these issues in the following subsections.

### 5.1.1 Algorithms and Study Subject Sizes

As stated in Chapter 2, TCP and TCS optimize for fault detection. As it is a priori unknown where to find faults and by which tests, it is a standard approach to use coverage as a surrogate for faults [208], i.e., to optimize or select tests for high coverage. This assumes that a test suite with high coverage, i.e., a test suite with a broad execution of the software is more likely to find faults compared to a more narrow test suite.

Based on related work, we identify two groups of algorithms that tackle TCP and TCS problems with coverage based approaches. The first group, *traditional algorithms*, contains algorithms that use only the set of available tests to propose the next test to execute. The second group, *overlap-aware algorithms*, contains algorithms that consider additionally the existing state, i.e., the set of already selected tests, as input for the next decision. For example, in terms of coverage, they consider the currently already covered code as input for the decision of the next test to select.

Traditional algorithms  
Overlap-aware algorithms

We show the difference by an example. Let tests A, B, C execute the sets of lines  $L_A$ ,  $L_B$ ,  $L_C$  with 100, 90, and 50 lines of code (LOC), respectively. A traditional algorithm might propose to run first A and then B (TCP), or select A and B (TCS). However, if  $L_B \subseteq L_A$ , and  $L_A \cap L_C = \emptyset$ , then the more efficient choice is A and C because they cover 150 LOC, but A and B only 100 LOC. This choice requires overlap-aware algorithms as these consider the sizes of intersections between  $L_A$ ,  $L_B$ , and  $L_C$ . Overlap-aware algorithms are also known in the literature as additional statement coverage algorithms, or additional greedy heuristics, among others (Section 5.1).

As we see later, several related work propose greedy algorithms similar to Section 3.3.6 for coverage-based TCS and TCP. However, there are also alternative heuristics proposed such as evolutionary algorithms and meta-heuristic optimizations [178, 257]. In our case, an overlap-aware greedy approach shows a good trade-off between precision and execution time.

Some approach do not only optimize for high coverage but also optimize for low test execution time. Such *time-aware* approaches assume that given two test suites with the same coverage, the one with the lower execution time is preferred. Previous work [261] claims that time-aware TCP makes no significant difference in terms of fault detection compared to standard TCP. Other work [77, 254] conclude that considering historical time data improves TCP techniques. It remains unclear whether the results for small projects can be generalized to large projects. For our project, feedback from developers is positive, but the results are not rigorously checked.

Time-aware

Work	Size	Term
[5]	5 classes to 22 classes	overlap-aware
[266]	53 testcases to 209 testcases	additional
[178]	374 LOC to 11 148 LOC	additional
[261]	500 LOC to 9 564 LOC	additional
[265]	2 kLOC to 80 kLOC	additional
[77]	7 kLOC to 80 kLOC	feedback technique
[82]	7.50 kLOC to 300 kLOC	additional
Our work	> 3.50 MLOC	overlap-aware

Table 5.1: Related work comparing overlap-aware and non-overlap-aware solvers for TCS or TCP. The column “Term” indicates which term is used for “overlap-aware”. Sizes in lines of code.

Several previous work compare overlap-aware algorithms to traditional ones, i.e., non-overlap-aware algorithms. Sometimes, the attribute “additional” is used instead of overlap-aware, e.g., “additional greedy”. Table 5.1 provides an overview of related work. In the following, we discuss details of related work and also indicate the terminology with: (term: “name”).

Zhang et al. [265] (term: “additional”) analyze the gap between additional and standard greedy approaches for software with 2 kLOC to 80 kLOC and propose several strategies to combine or outperform both greedy approaches. The advantage of overlap-aware algorithms is an indirect conclusion.

Li et al. [178] (term: “additional Greedy”) compare greedy, metaheuristic, and evolutionary search algorithms for six programs, ranging from 374 LOC to 11 148 LOC. Their results show the strengths of the overlap-aware greedy approach compared to the standard greedy. In terms of execution time, they conclude that the overlap-aware variant should be used.

Elbaum et al. [82] (term: “additional statement coverage”) compare multiple prioritization techniques by controlled experiments and case studies (sizes: 7 451 LOC, 9 153 LOC, 300 kLOC). They conclude that “techniques with feedback (addtl)” produce better results for two programs.

Zhang et al. [266] (term: “additional”) evaluate linear programming for time-aware test-case prioritization on two programs with 53 test cases (2 s execution time) and 209 testcases (6 s execution time). Their results indicate that additional techniques are superior to other techniques.

Alspaugh et al. [5] (term: “overlap-ware”) compare an overlap-aware greedy algorithm with different solvers for the standard knapsack problem

(greedy in multiple variants, dynamic programming, core, random) for two programs with 5 classes and 22 classes. They conclude that “overlap-aware solver achieves the highest overall coverage for each testing time constraint”, but suggest studies on larger applications.

Do et al. [77] (term: “feedback techniques”) compare “no order”, “random order”, the standard and overlap-aware variants greedy approaches, and Bayesian network algorithms for programs with 7 kLOC to 80 kLOC.

You et al. [261] (term: “additional”) compare random, greedy for coverage, time-aware greedy, and time-aware integer linear programming. All three algorithmic variants in both versions overlap-aware and standard. The analysis includes eight software projects, seven with up to 500 LOC and one with 9 564 LOC, total test execution times up to 33 s.

Walcott et al. [254] compare a genetic algorithm with “normal” order and reverse order. They also use very small study subjects. They do not compare to a standard greedy approach.

In conclusion, there is a wide range of previous work on coverage-based TCP and TCS. However, the study subjects are typically rather small in terms of lines of code. Such studies may lack generalizability for large projects. Therefore, we compare several proposed approaches in Section 5.3.

### 5.1.2 Shared Coverage, Randomness, and Flakiness

Our large study subject shows several phenomena rarely addressed in prior work. One of them is that coverage for system tests contains a large share of code from *shared functionality* such as startup code or internal libraries. This makes it harder to distinguish tests based on coverage and poses problems for, e.g., change-centric testing. To the best of our knowledge, this problem is not studied so far. We assume that this phenomenon becomes visible only in larger software projects. We address this issue in Section 5.5.

Shared functionality

Partly related to the previous issue, the coverage data of our study subject shows a high redundancy of (line) coverage data. This means that e.g., either all or none of the lines in a group of lines are covered by a test. Such redundancy offers opportunities to significantly reduce the size of coverage data. Epitropakis et al. propose an algorithm to filter such redundancy in line coverage [85]. We analyze and extend this algorithm in Section 3.3.8.

High redundancy of coverage data

Finally, we also observe *randomness* in terms of test results (i.e., flaky tests [181]) as well as in terms of covered code. Such randomness occurs frequently for our study subject and is a threat to the effectiveness of prior approaches. We further discuss these issues in Section 5.6.2.

Randomness

### 5.1.3 Conclusion

As shown by our discussion, there are several reasons why approaches proposed for test cost reduction by previous work may not be effective for large projects. In the rest of this chapter, we propose an approach for test cost reduction adapted for the specific characteristics of a large project, we will investigate the effectiveness of multiple coverage-based algorithms for large projects and we will further analyze the specific issues related to randomness in large projects. Finally, we also discuss how our work is affected by various issues found in large projects.

## 5.2 An Economic Approach for Test Cost Reduction

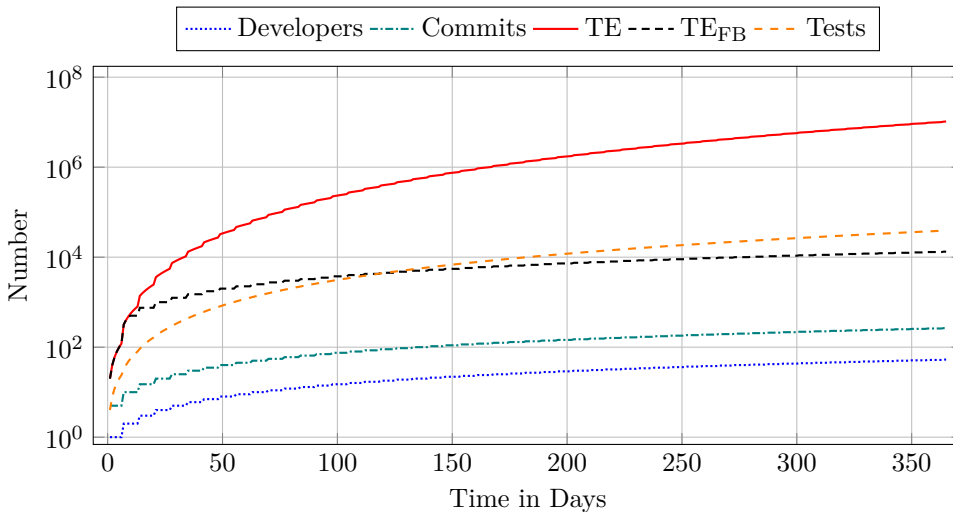
Extensive testing in large projects can lead to tremendous test costs with superlinear growth over time. Researchers have proposed several techniques to tackle this problem [208]. However, the practical effects of these techniques on the asymptotic behavior of test costs growth in large industrial software projects remain poorly characterized.

We introduce and analyze a *fixed time budget for test executions* for SAP HANA. Our approach assigns a global fixed time budget to several components of SAP HANA that are development units within SAP HANA. Each component can only execute tests within its budget, which can change only by transferring time budget between components. This limits the number of test executions for each test run to a constant, thus reducing the asymptotic growth of test costs.

Budget transfers and test optimizations adhere to balances between value and costs, thus creating an *economic environment for test case selection and reduction*. Specifically, this creates incentives to remove unnecessary tests and to optimize test execution times.

For SAP HANA, our approach leads to effective test case selection and reduction, and reduces test execution times by 105 years in a time frame of four months with a negligible effect on quality. The trade-off between test execution time savings and failure detection is 1.83 years/failure.

### 5.2.1 Introduction



Fixed time budget for test executions

Economic environment for test case selection and reduction

Figure 5.1: Model of the test execution growth in logarithmic scale. The number of developers (*Developers*) increases by one every seven days. Each developer increases the number of existing tests (*Tests*) by four per day and the number of commits (*Commits*) by five code changes per day, each initiating a test run. The number of test executions (*TE*) is equal to  $Tests \times Commits$  and grows superlinearly over time due to the linear increases of *Tests* and *Commits*. A fixed time budget for test executions effectively limits the number of executed tests to a constant  $c$ . Therefore, the number of test executions with a fixed budget ( $TE_{FB}$ ) is equal to  $c \times Commits$  and grows linearly over time.

As discussed in Chapter 1, it is important to reduce test costs in large projects. In large projects, the time spent on test execution can contribute substantially to the test costs. These test execution times increase superlinearly over time due to the typical characteristics of large software projects that affect the number of test executions  $TE$ .  $TE$  depends on two factors:

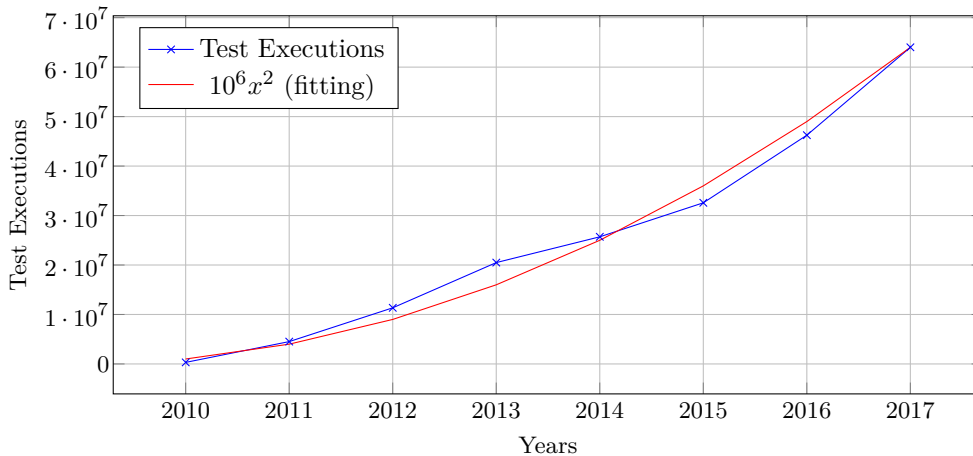
1. The number of tests  $T$  that are executed, and
2. The frequency  $F$  the tests are executed.

Test executions increase superlinearly over time.

$T$  increases linearly over time as developers add more tests but rarely delete tests.  $F$  increases linearly over time because a successful project has an increasing number of developers making changes to the code. Each change initiates a new test run and thus increases  $F$ . The multiplication of the two linear factors  $T$  and  $F$  lead to a superlinear increase for  $TE$  over time. The model of Fig. 5.1 demonstrates the superlinear increase of  $TE$ . After one year, 53 developers have produced 39 004 tests, and create 265 code changes per day, resulting in more than 10 million test executions per day. This model already shows that even with relatively small input variables, the resulting test costs are large.

In addition to this general issue, there are at least two other aspects with a considerable effect on test costs. For long term projects as SAP HANA, there exist *multiple product versions* for their customers that require parallel test executions due to differences in features and code state. The leads to an additional constant or potential linear factor of test executions. Flaky tests increase costs for all test stages due to additional required inspections and reruns for misleading test results (See Section 5.6.1).

Overall, we expect that test executions and test costs increase superlinearly in large projects. For SAP HANA, Fig. 5.2 confirms the superlinear growth for test executions between 2010 and 2017. Section 5.2.5 highlights other work that confirm the superlinear growth.



Multiple product versions

Figure 5.2: SAP HANA test executions with a quadratic curve fit. The dataset contains all centralized automated test executions since 2010.

For the development of SAP HANA, SAP adopted a fixed time budget for test executions to reduce test costs. The fixed time budget implements TCS and TSR techniques (see Chapter 2). Periodic test runs with all tests in later testing stages prevent the potential loss of quality. We analyze the effects of this approach after four months. Our contributions are as follows:

- An approach to reduce test costs that changes the superlinearly increase over time to a linear increase.
- An analysis of this approach for a large industrial project.

Section 5.2.3 introduces the testing environment of SAP HANA. We describe our approach in detail in Section 5.2.3. Section 5.2.4 contains our research questions, results and discussions. Section 5.2.5 briefly highlights related work. We conclude in Section 5.2.6.



### 5.2.2 Testing of SAP HANA

Section 2.2 provides a general introduction to SAP HANA. Here, we focus on the practical testing process.

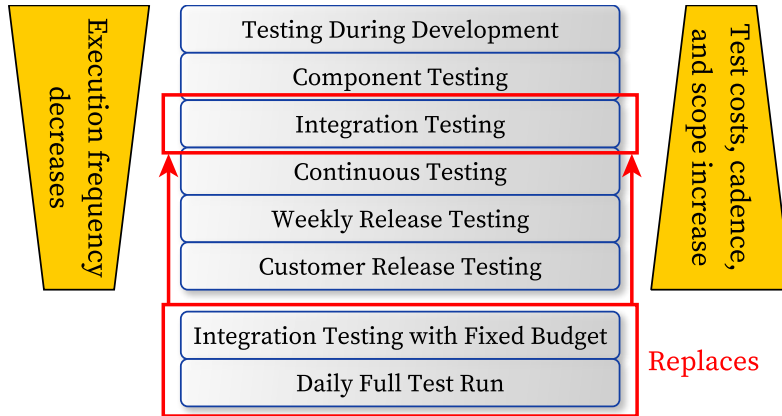


Figure 5.3: Simplified summary of test stages for SAP HANA. For the implementation of our fixed test budget approach, we replace the (full) integration testing stage with two new stages. The two new stages consist of a test run with a fixed time budget for each code change and a daily full test run. The fixed time budget for test executions reduces the growth of the costs at the integration testing stage. The daily test run ensures the same quality level as before the replacement.

Testing of SAP HANA contains several stages as shown by Fig. 5.3 and involves a *continuous build environment* (CB). At the first stage, developers execute tests during software development on their own workstation. This can include manual tests, a subset of test suites from later testing stages or developer-specific test suites. The second stage incorporates the CB to automatically execute integration tests of a component. This stage executes a set of tests decided by a component, i.e., a set of functionalities.

Stage three includes the integration and regression testing for all SAP HANA components and causes the majority of the test costs because of the superlinear cost increase explained in Section 5.2.1. Fig. 5.2 shows the current trend of test executions for SAP HANA. Stage three implements pre-commit testing, i.e., a software change is only accepted and merged into the repository if it passes all tests. Even more, in the case a failure is found, each adaption may require to re-execute all tests.

Within testing stage four, the CB system runs different combinations and configurations of tests (test profiles) after code changes have been submitted to the repository. The execution frequency for these profiles reaches from several times a day to once per release. Stage four includes long-running tests, randomized and stress testing, or recovery testing like out of memory or crash tests. The test profile for releases contains over 900 000 single test cases and would run for 23 days if executed sequentially on an average server with 40 CPU cores, 3 GHz frequency, and 256 GiB of memory. Although SAP utilizes a large hardware pool of test servers in parallel mode, the execution time for a test run is rather large with several hours. Within the test suites of a database, it is not uncommon to have long-running, complex and distributed queries to test database functionality. Performance and load tests contribute to long-running tests as well.

Whereas the stages one to four focus on SAP HANA in an isolated environment, stage five and six perform end-to-end testing, i.e., testing the same scenarios as user would use the final product. These later stages contain automatic, but also manual tests that are performed by separate teams within the company and thus decrease testing biases.

Testing stages

About one million test cases

### 5.2.3 Fixed Test Budget Approach

To limit the execution time of the integration stage, we introduced a fixed time budget for tests where each component of SAP HANA receives a budget of test execution time. The component team can select the tests to run by their own criteria if the cumulative execution time fits into the budget. Time budgets can be shifted between components to cope with changing priorities and quality requirements within SAP HANA development. These budget redistributions allow the component team members to focus resources in areas of special interest. In addition, SAP also expected that the fixed time budget would benefit them in the following ways:

Components can shift budget.

1. Tests with low bug-finding abilities will move to later testing stages (by manual or automatic techniques).
2. Execution times for tests will be analyzed and optimized, especially for long-running and distributed tests.
3. Test scope will improve. A test suite for a component should only include necessary tests for the quality assurance of this component. This could require modifications of the test suite or the component structure.

To implement our approach, we need to specify the total time budget and the distribution to all components. This task is particularly difficult in the context of SAP HANA. We first evaluated distributions based on empirical data about test executions. However, this requires a complex methodology for the collection process as it has to account for several practical challenges. For example, a long period would not include recent changes, a short period would not include long term trends and a weighted mix would be difficult to understand by all developers. In addition, test ownership is not well defined if multiple components use the same tests in their component test suites.

We decided to devise a simple model that accounts for the team sizes to output the budget cost. Albeit simple, this model is easy to explain to Stakeholders. The simple model might have inaccuracies, but after the initial distribution, the possibility to transfer test budgets between different components allows a flexible redistribution of time budgets.

Initial budget based on component team sizes

The component teams were responsible for the process of test selection and reduction to fulfill the test budget constraint for their respective component. To support the component team members with test case selection and test suite reduction, we created an *extensive statistics* of execution times and failure rates for the tests contained in the integration stage over a time frame of 6 months. For example, we found that 17% of all tests had a failure rate less than 0.01% and 11% indicated no failure at all for the observation time. The criteria for test selection and reduction which were used by the component teams were not evaluated. We expect the teams used a mix of domain knowledge and empirical data. A future study could investigate if these criteria can be utilized for an automated approach.

Extensive statistics

The introduction of a fixed time budget for integration tests can reduce the test quality because only a subset of tests is executed. To avoid quality degradation in later testing stages, a *daily test run* executes the original set of integration tests. This daily test run effectively moves the complete test run from pre-commit to post-commit testing.

Daily test run

Altogether, the introduction of the fixed time budget for the integration test stage changes the asymptotic behavior of the number of test executions  $TE$  over time. Before the introduction,  $TE$  increased superlinearly over time, as discussed in Section 5.2.1. After the introduction,  $TE$  increases only linearly over time on both new integration test stages. For the integration testing with a fixed budget, the test budget limits the maximum execution time of tests.  $TE$  depends on a fixed maximum time (i.e., a constant) and a variable number of commits over time, resulting in a linear growth for  $TE$  over time. For the daily full test run, the frequency is fixed.  $TE$  depends on a variable number of tests over time and a fixed execution frequency (i.e., a constant), resulting in linear growth of  $TE$  over time.

The fixed time budget reduces the growth of test executions from super-linear to a linear.

#### 5.2.4 Results

We investigated the following research questions (RQ) in the context of the large-scale software project SAP HANA:

- RQ6 How does the distribution of the fixed test budget for test executions change over time?
- RQ7 How did execution times for integration tests change after the introduction of the fixed time budget?
- RQ8 How is the testing quality affected by the introduction of the fixed test budget for test executions in terms of failures that pass the reduced test suite, but appear in the full integration test suite?

RQ6 analyzes whether component teams redistributed test budget between components. This indicates whether the economic aspect of benefits and costs lead to re-prioritization of testing efforts. RQ7 investigates the direct effects on developer processes and hardware costs and analyzes whether test execution times were reduced. RQ8 analyzes the impact of the fixed time budget on the quality of code change testing. A significant increase in the number of defects in later testing stages could imply that the fixed test budget for code changes has a strong negative impact on the product quality. This would question the usefulness of the approach. On the other hand, a low increase in defects in later testing stages (or none at all) would indicate that the trade-off between test execution times and quality is acceptable. In addition, we quantify the trade-off between execution time and failures in later test stages in terms of time saved per failure.

##### 5.2.4.1 RQ6 Budget Redistribution

Fig. 5.4 visualizes how the time budget of each component changed between  $t_0$  (introduction of the fixed time budget for test executions, 2017-08) and  $t_1$  (2017-12). Unfortunately, for budget redistributions, we do not have the source and target components for all changes. Therefore, we cannot show an alluvial diagram to illustrate the fine-grained changes. We rescaled the unchanged budget part because it does not provide further information. The percentage of the unchanged part differs in  $t_0$  and  $t_1$  due to an increase in the total time budget. The total time budget increased due to readjustments after the initial introduction. In total, the time budget increased for 18 components and decreased for 27 components between  $t_0$  and  $t_1$ .

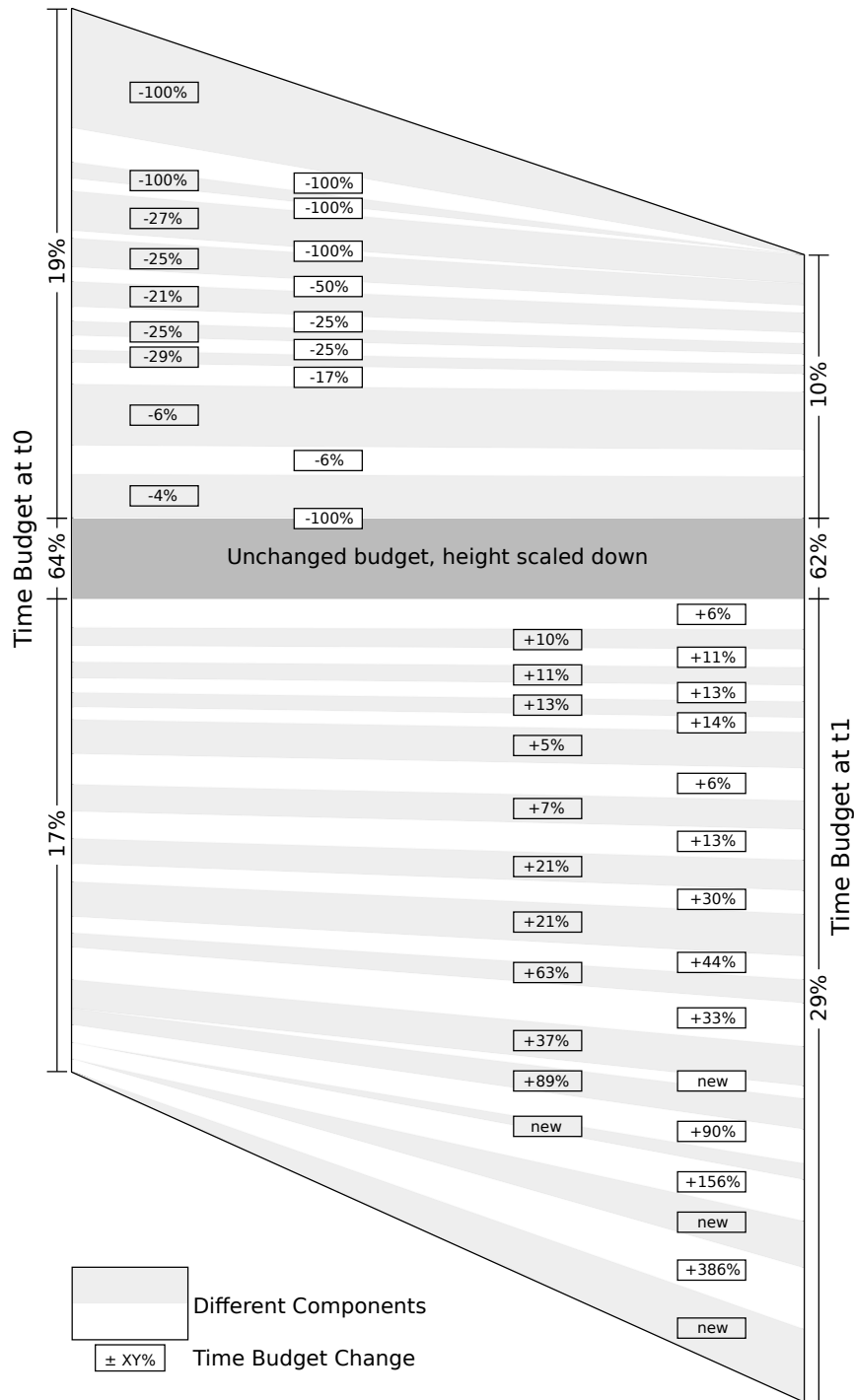


Figure 5.4: Changes in time budgets for each component over time from  $t_0$  (introduction of the time budget, 2017-08) to  $t_1$  (2017-12).

Fig. 5.4 shows that time budget transfers occur in practice to optimize benefits and costs. This is an indicator that the expected economic effect exists in practice. Further investigations of single cases show that there are multiple reasons such as:

- SAP split some components to narrow down their test scope, which increased test efficiency.
- Component team members adapted their requirements and free test budget was reallocated to other components.
- SAP identified and improved or removed tests with long execution times.
- SAP reallocated tests with long execution times to later stages of the testing process with decreased execution frequency.

Overall, the distribution of the fixed test budget changed over time which indicates that component teams trade test budgets. Our analysis indicates that this can lead to positive effects for the state of test suites due to an incentive to maintain the test suite from a cost and benefit perspective.

*Answer RQ6*

The changes over time for the distribution of test budgets reflect different priorities and increased awareness of costs associated to test activities. Overall, economic effects affect the management of test suites.

#### 5.2.4.2 RQ7 Changes in Test Execution Times

Within the testing process of SAP HANA (Section 5.2.2), the fixed test budget affects only integration testing for the main product line. Other integration stages have different policies for test selection. We analyze the test cost changes in terms of execution times for the integration testing to the main product line and for all integration testing.

The fixed test budget defines the maximum threshold of execution time for integration tests to the main product line. At the point of the introduction  $t_0$  (August 2017), the difference between budget and actual execution times was a factor of 4. After  $t_0$ , the actual execution times are nearly equal to the budget with some deviation. Deviations occur because of re-adjustments, fluctuations in test execution time, and variations in infrastructure workload because several projects share the available resources. Fig. 5.5 shows the execution time statistics for multiple product lines.  $M$  represents the main product line. Fig. 5.5 shows that the total execution time for  $M$  decreased from August to September by a factor of 2. Based on further analysis, we conclude that the reasons for the smaller than expected reduction are test overheads like compile times, binary redistribution, database install and setup times, and test setup times.

Fig. 5.5 shows that the fixed time budget for test executions saved hardware resources, but the figure also shows that additional product versions counterbalance the savings. As a result of this study, SAP engineers introduced the fixed time budgets also for other product lines.

The overall average execution times for all integration tests exhibit a more complex pattern. Fig. 5.6 visualizes the state before and after the introduction of the fixed test budget. Due to confidentiality reasons, we cannot provide the exact numbers. Therefore, Fig. 5.6 only shows the scaled

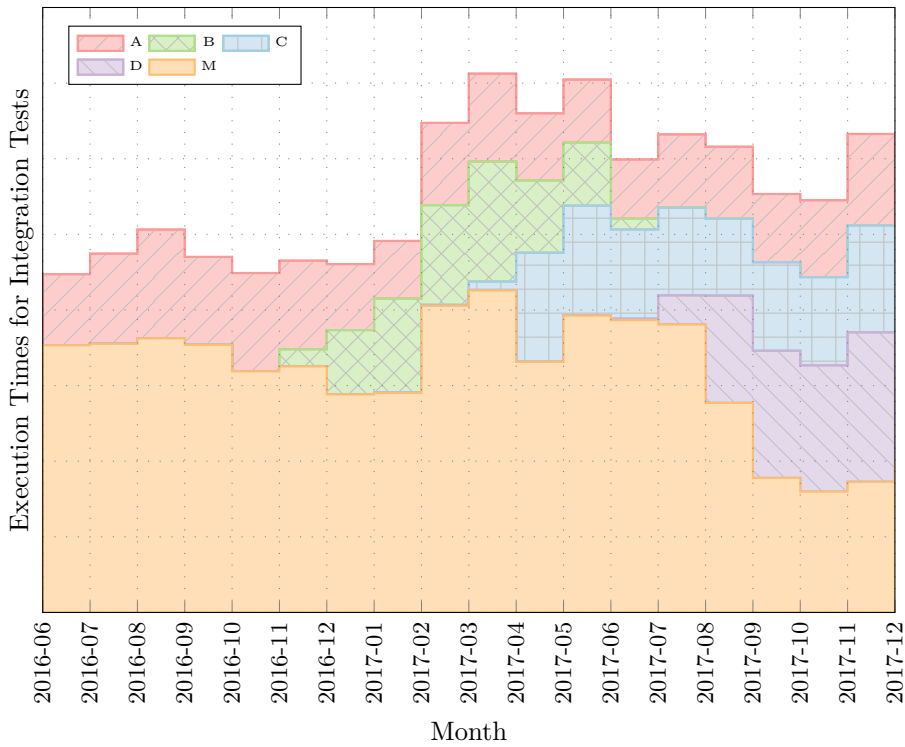


Figure 5.5: Stacked area chart with cumulative execution times for tests for product lines *M* (main), *A*, *B*, *C*, *D*. The test execution times for *M* decreased in August 2017 due to the introduction of the fixed time budget for test executions.

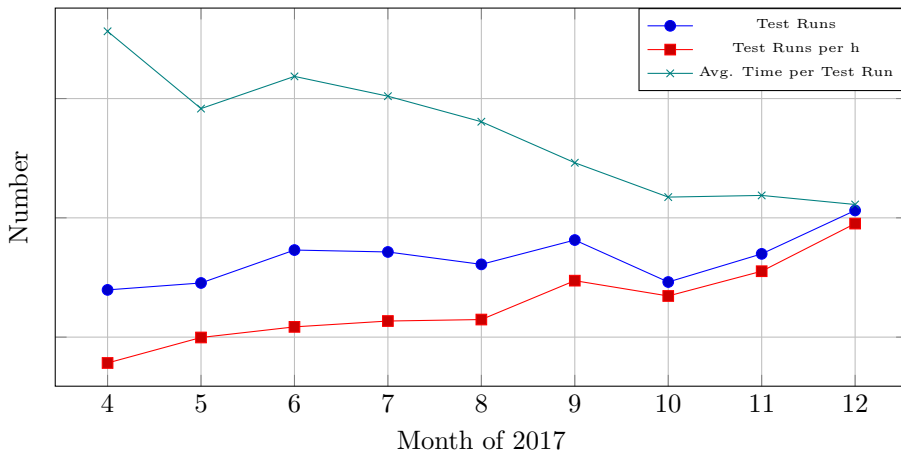


Figure 5.6: Execution times for integration tests and test runs over time. Execution times decrease and test runs increase and therefore test runs per hour increases.

data. The average test execution times  $R_{avg}$  decreased after the introduction of the fixed test budget in August 2017. This leads to an increase in test runs  $TR$ , which is explained by hardware limitations.  $TR$  multiplied with  $R_{avg}$  cannot exceed the capacities of the existing hardware. Therefore, if the product is fixed,  $TR$  must increase if  $R_{avg}$  decreases. The increase in  $TR$  and decrease in  $R_{avg}$  result in an increase by factor two for the quotient test runs per hour which represents efficiency.

Overall, Fig. 5.5 and Fig. 5.6 indicate that the fixed test budget has a positive effect on test execution times and the efficiency. It is unclear whether the current trend continues and if yes, why. We would expect that after the introduction of the test budget, there would be a sharp decline of a factor 4 in the test execution time which would then stay nearly constant. Our data shows that this factor is not fully reached. We assume that previously, developers reduced the testing to cope with waiting times. The ongoing changes could indicate that component teams are continuously improving their tests and therefore optimizing their test budgets.

— Answer RQ7 —

The fixed test budget reduced the average execution times of test runs by a factor of 2.5 and the total time spent on testing by a factor of 2.

#### 5.2.4.3 RQ8 Changes in Quality

We analyze all internal bug reports for valid test failures and counted them over time (this excludes flaky test results). Fig. 5.7 shows the results. Statistically, there are on average 3.2 failures per week. This number is remarkably low for a large project such as SAP HANA.

In addition, we compare the results from Section 5.2.4.2 (changes in execution times) and Section 5.2.4.3 (changes in quality) to quantify the trade-off between execution time savings and failures in later test stages in terms of time saved per failure. We estimate time savings  $R_s$  by interpolating of test execution times without the fixed time budget  $R_i$  and subtracting of the actual test execution time  $R_a$ . In Fig. 5.5, the last known test execution time  $R$  before the introduction of the fixed test budget is shown by the datapoint 2017-08 for  $M$ . We multiply  $R$  by 5 for 5 months. We can determine  $R_a$  by the sum over all actual test execution times in  $M$  for the last 5 months. Now, we can calculate the time savings  $R_s = R_i - R_a$ .

Finally, we obtain the result that SAP saved 104.50 years of test execution time due to the fixed time budget. With the average of 3.2 failures per week indicated by Fig. 5.7, we can calculate a quotient of 1.83 years/failure. This implies that SAP traded in average a test execution time of 1.83 years against one additional failure in a later test stage. This number only provides an approximation of the trade-off and does not necessarily reflect the reality precisely due to the interpolation and the low number of data points.

— Answer RQ7 —

On average, 3.2 failures per week are only detected by the next testing stage. This implies that, on average, a test execution time reduction of 1.83 years is achieved for one additional failure in the next testing stage.

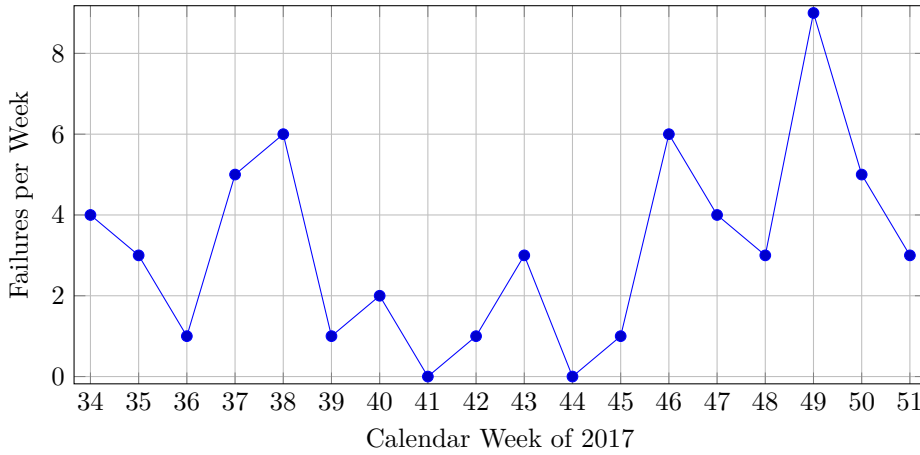


Figure 5.7: Failures undetected by the reduced test suite but within the full test run. Aggregated by calendar week due to the different behavior of weekdays and weekends. Average: 3.2 failures per week.

### 5.2.5 Related Work

Yoo et al. provide a survey about test case selection and test suite reduction [257]. A recent systematic literature review about test case selection by Kazami et al. highlights different trends and results in this area [150]. Blondeau et al. analyze test case selection within several industrial projects [27]. Their work focuses on change-based testing, our approach does not depend on dependencies between changes and tests.

Our approach shares characteristics with risk-based testing. Although, as explained in Section 5.2.3, we mitigate risks by a daily full test run. Felderer and Schieferdecker provide a taxonomy of risk-based testing [89]. Risk-based test planning subsumes techniques that select or prioritize test cases based on a risk analysis of their costs and value.

Memon et al. study the same problem of polynomial growth for test executions at Google [191]. The approach used by Google utilizes a manual dependency list for each product to collect all required test runs for a source code change. The test framework collects these required test runs over a period of four hours and runs each required test only once. An automatic backward analysis identifies failure-introducing commits. This approach reduces the linear growth of test executions by commit frequency to a constant factor due to the fixed number of executions per day. However, our fixed test budget creates additional incentives for developers to reduce and improve existing test suites. Additionally, our multi-stage separation enables faster individual feedback times for developers.

### 5.2.6 Conclusions

We described an approach to use a fixed time budget for testing that is adapted by SAP to limit the test execution growth. We analyzed the initial effects on execution times and quality in terms of failures that pass the reduced test suite, and we analyzed the economic effects. Our analysis indicates that there are positive effects on execution times and test suite efficiency, while the negative effects on quality are low.

Although the current observations show possible trends, we cannot conclude statistical significance due to limited data. Further work would require



an observation time of at least one year to improve the statistical significance. One year would cover periodic variations, which can be observed for example at the end of the year due to vacations or directly before and after releases of new major software versions.

Based on internal discussions, the results of our analysis are plausible and the impact on test execution times and failure detection is reasonable. Surveys or questionnaires with developers would provide further insight on the impact of the fixed time budget on the individual developer. These findings could support further refinements of our approach. As a direct consequence of this work, the fixed time budget for test executions is introduced for parallel product lines, as explained in Section 5.2.4.2.

### 5.3 Test Case Selection and Prioritization

As we have shown in Section 5.1, existing evaluations of overlap-aware algorithms were conducted on comparatively small projects. Our study for SAP HANA in Section 5.3.2 shows that overlap-aware algorithms provide considerable improvements in terms of potential time savings. This high impact is related to the large amount of shared coverage for the test suites of SAP HANA, which can be less pronounced in other projects. Considering the fast execution times, we argue that only overlap-aware algorithms should be used for coverage-based TCP and TCS optimization problems.

Overlap-aware algorithms should be preferred.

#### 5.3.1 Approach

**Test case selection.** We focus first on the issue of reducing cumulative time required for test runs and consider two alternative problem formulations: (i) Given a fixed time budget for the execution of tests we attempt to find a subset of tests with maximum coverage, and (ii) we try to find a subset of tests with minimal execution time which cumulatively achieves the best possible coverage (understood as cumulative coverage by all available tests).

More formally, let  $R$  be the set of all available coverage files  $c_1, \dots, c_n$  (with  $c_i$  corresponding to a test  $i$ ) that contain the set of covered lines for all source code files. For  $P \subseteq R$  let  $time(P)$  denote the cumulative time to execute all tests specified by  $P$ . Furthermore, let  $T$  be a time budget (threshold), and assume the add (+) operation according to Section 3.3.2. Then the test case selection (TCS) problem can be expressed in either one of the following dual formulations:

TCSa Maximize  $|\sum_{c \in P} c|$  with  $time(P) \leq T$  and  $P \subseteq R$ ,  
 TCSb Minimize  $time(P)$  with  $\sum_{c \in R} c = \sum_{c \in P} c$ ,  $P \subseteq R$ .

	$t_1$ (16 s)	$t_2$ (15 s)	$t_3$ (15 s)
L1	x	x	
L2	x	x	
L3	x		x
L4		x	x
L5	x		x

Table 5.2: Example coverage data.  $x$  denotes that a test executes (“hits”) a line.

In both cases, we optimize over the set  $P \subseteq R$  which translates to finding an optimal set of tests. Table 5.2 shows an example. For **TCSa**, a subset with maximal coverage for a time budget of 15 s is  $\{t_2\}$ , for a time budget of 20 s it is  $\{t_1\}$ , and it is  $\{t_2, t_3\}$  for a time budget of 30 s. For **TCSb**, the subset with full coverage and minimal time is  $\{t_2, t_3\}$ .

**TCSa** is a variation of the general 0/1 knapsack problem [151]:

$$\text{maximize } \left| \sum_{j=1}^n p_j \times x_j \right| \quad (5.1)$$

$$\text{subject to } \sum_{j=1}^n w_j x_j \leq T \quad (5.2)$$

where:

- $x_j \in \{0, 1\}$ ,  $j = 1, \dots, n$
- $T$ : budget on weights (i.e., time budget)
- $M : \{1, \dots, n\}$  set of items (i.e., tests)
- $p_j$ : "profit" of item  $j \in M$  (i.e., coverage file)
- $w_j$ : weight of item  $j \in M$  (i.e., execution time).

In Eq. (5.1) we use (with  $P = \{p_1, \dots, p_k\}$ ):

- $p_i + p_j$ : as defined in Section 3.3.2
- $p_i \times x_i : \emptyset$  if  $x_i = 0$ ,  $p_i$  otherwise ( $x_i \in \{0, 1\}$ ).

Note that  $x_j$  indicates whether a test  $j$  was selected ( $x_j = 1$ ) or not ( $x_j = 0$ ). In Eq. (5.1), we can see that **TCSa** is similar to the 0/1 knapsack problem, but the major difference is the overlap of coverage elements  $p_j$ .

Problem **TCSb** is known as the *Weighted Set Cover Problem* (WSCP). Section 3.3.6.1 discusses the WSCP and algorithms to solve it. Similarly, we address problem **TCSa** and describe an algorithm to solve it in Section 3.3.7.1. Dynamic programming (DP) is a typical textbook approach for knapsack problems [151]. However, DP assumes that a solution for a large problem instance consists of solutions for smaller instances and solutions with the same weight and profit are equal. This property does not hold for sets, i.e., two solutions with the same weight and profit can lead to different results if re-used later because the set of existing elements in the current solution affects the weight of newly added items. Therefore, such DP approaches as used in related work [5] should not be used to solve the WSCP.

Approaches based on dynamic programming might not be suitable.

Note that both test case selection strategies are not safe, i.e., they can omit test cases that would otherwise have detected faults. This must be considered during a risk analysis of the costs and benefits. Related work provides more information about safe selection strategies [257].

**Test case prioritization (TCP)**. We can use the solution set from TCS as input for the TCP problem. This requires test reordering so that tests with the best ratio of code coverage over execution time run first. This is also called *time-aware test suite prioritization* [254, 261].

Note that in the general case we can only find (any of) the best order if the exact amount of time (or tests) is fixed. We illustrate this with the example shown in Table 5.2. We run  $t_1$  with 4 s/line first. This is the best solution for a solution set with only one element, because  $t_2$  and  $t_3$  have 5 s/line. The next test to add to the existing solution set must be either  $t_2$  or  $t_3$  which gives us a total execution time of 31 s for 5 lines. But we have done overall better if we choose  $\{t_2, t_3\}$  with a total execution time of 30 s for 5 lines. This example shows that we cannot find the best solution for all cases (without a fixed time budget or a fixed amount of tests).

### 5.3.2 Evaluation

We investigate the following research question (RQ) for SAP HANA:

RQ9 Does an overlap-aware heuristic solver for TCS produce better results than non-overlap-aware solvers?

To answer RQ9, we compare the effectiveness of the overlap-aware greedy (OAG) approach outlined in Section 5.3.1 against a standard greedy (SG) implementation. We apply all algorithms on the same coverage data with different time budgets (increased in 1h steps). For each time budget, we get a solution in terms of the total number of covered lines (“sum of lines hit”) — the higher the sum, the better. The execution time of all algorithms is less than 10 seconds, therefore we do not report these times.

Fig. 5.8 shows the results for one coverage run, while Table 5.3 exhibits the results of different coverage runs over one year. OAG converges to high coverage and reaches the maximum significantly faster than SG. The evaluation shows that SG has similar behavior to a random shuffle and in some cases, it is even worse than a random guess. OAG is significantly better, with a factor of 1.40 up to 1.50 for a time budget of up to 30 hours.

Coverage Run	Greedy Variant	Req. Time Budget (hours) for		
		90 %	99 %	100 %
2015-11-15	standard	74	137	137
	overlap-aware	19	57	123
2016-05-19	standard	110	182	191
	overlap-aware	25	70	173
2016-10-25	standard	108	193	219
	overlap-aware	24	72	196

Table 5.3: Required time budgets for BMCP to reach different percentages of total coverage (cases span one year).

A comparison between OAG and SG for TCP leads to nearly identical results. We also evaluated the impact of parallelization as described in Section 5.3.1 with parallelization factors ranging from  $p = 1$  up to  $p = 50$ . The relative savings are almost identical, the differences are less than 0.01 %.

#### Answer RQ9

Compared to a non-overlap-aware approach, an overlap-aware heuristic solver is substantially better, with a factor of 1.40 up to 1.50 for a time budget of up to 30 hours. A non-overlap-aware approach performs similarly to an approach based on random decisions.

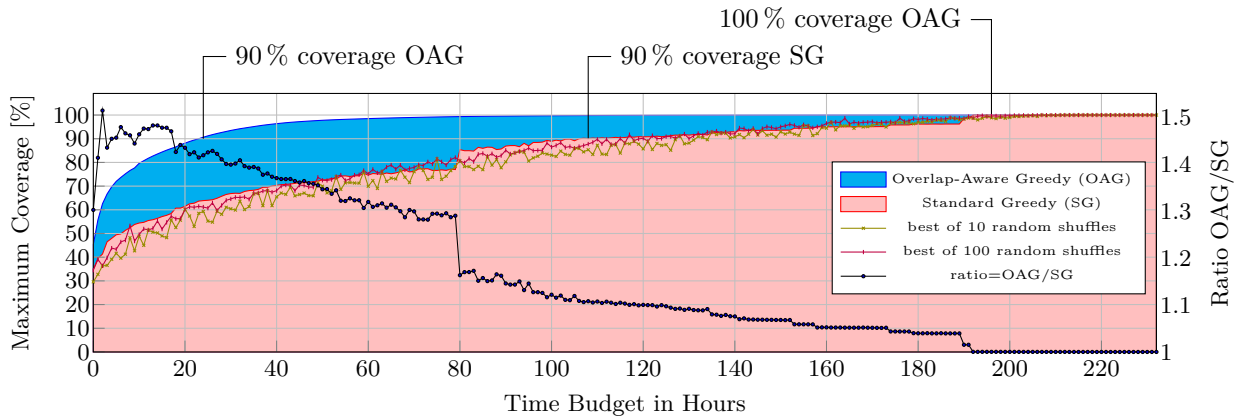


Figure 5.8: Exemplary comparison of different algorithms for the maximum budgeted coverage problem. Higher is better.

### 5.4 Size of Coverage Data

A problem less pronounced in the literature is the size of the coverage data. In the case of SAP HANA, the cumulative size of coverage data for one coverage run has reached 130 GB in 2016 and 162 GB in 2019. The evolution of this size over one year is shown in Fig. 5.9. For more advanced analysis such as the discovery of trends and comparative studies, several months worth of data is stored. The total storage size is 14 TB as of 2016.

Over 150 GB data for a single coverage run

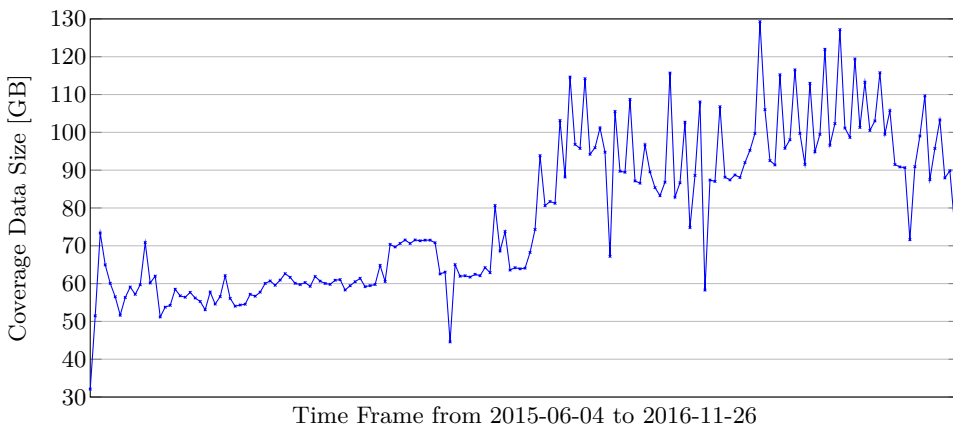


Figure 5.9: Growth of data size for coverage runs over one year.

Despite the storage requirements, processing of such data requires higher computational capacities. Analyzing the results of a single coverage run can require a matrix with more than  $10^9$  entries (with an observed density of 9% to 12%). Therefore, we investigate methods to reduce the data size.

#### 5.4.1 Approach

We apply several techniques to reduce the coverage data size:

1. Remove uninteresting data such as lines that were not executed.
2. Use space-efficient data structures such as a bitset.
3. Use dictionary compression to replace long strings with short identifiers.
4. Remove redundancy by exploiting the inherent structure of coverage data.

Technique 1 is rather simple. In all cases where we only require information about executed lines, we can discard all other lines. We can later restore them if required. For technique 3, we utilize a dictionary that translates long words, filenames in our case, into short identifier, such as a number [201]. In our case, the dictionary  $d$  is a mapping  $d : S \rightarrow \mathbb{N}_0$  where  $S$  represents any string. Each time we process a string  $s$ , we either look up the number in the dictionary  $d(s)$  if the key exists or, if not, we assign  $s$  the next smallest integer that is not used, i.e.,  $d(s) = \max(d.values) + 1$ . We then store the dictionary for later usage and use only the identifier instead of the filenames. Technique 2 uses  $n$  bits to encode for  $n$  lines whether each line was executed or not. For example, a single 64 bit integer can then represent 64 lines, see Section 3.2.5 for implementation details. For technique 4, we create equivalence classes (EC) over coverage sets and use only a single representative line for each EC. The last technique requires further discussion.

When we execute a software, a computer interprets and executes a sequence of instructions  $S_{instr}$ . Simplified, the CPU executes the instruction at position IP in  $S_{instr}$ , increases IP, and repeats these steps until a termination condition. We call IP the instruction pointer. IP increases linearly until the program ends or a *control flow instruction* (CFI) modifies IP. For example, statements such as if/then/else, loops, or function calls translate to CFI.

Let  $SI_D$  be a subsequence of consecutive instructions in  $S_{instr}$  where  $SI_D$  contains no CFI. A CPU will either execute all instructions in  $SI_D$  or none of the instructions in  $SI_D$ . Therefore, given coverage data for the instructions in  $SI_D$ , the coverage state of the first instruction (i.e., executed or not executed) determines the coverage state of all other instructions in  $SI_D$ . Hence, we only need the coverage state for one instruction in  $SI_D$ , the coverage data for all other instructions is redundant. Fig. 5.10 shows an example where either each line in the body of the `calc` function is executed or none of the lines are executed. Furthermore, the coverage state of the `calc` function also determines the coverage states of the `sum` and `mul` function (assuming no other functions call `sum` or `mul`). This shows that redundancies can even span over multiple subsequences of instructions.

Such redundancies in coverage data occur quite often, because control flow instructions appear only in comparatively few cases. Akshintala et al. report for x86-64 in C/C++ programs that only 22% of all instructions are control flow instructions (such as `CALL`, `JE`, `JMP`, `CMP`, `JNE`, `TEST`). Removing `CALL` reduces the percentage to 16% [4].

Furthermore, even in the cases where a CFI is executed, there might still exist redundancies in coverage data:

- A control flow statement can result in the same execution flow for all executions and therefore generate redundant coverage data. For example, an if statement always executes the else branch. Conceptually, we can remove the instructions that were never executed to obtain a consecutive sequence of instructions without CFI.
- A function call may always follow the same control flow and therefore generate redundant coverage data. Conceptually, we can integrate the function body at the current place of our instruction sequence to obtain a consecutive sequence of instructions without CFI.

Control flow instruction

Coverage of a single instruction determines coverage for a block.

```

1 int sum(int a, int b) {
2     int sum = a + b;
3     return sum;
4 }
5 int mul(int a, int b) {
6     int mul = a * b;
7     return mul;
8 }
9 int calc(int a, int b) {
10    int x = sum(a, b);
11    int y = sum(a, -b);
12    int binom = mul(x, y);
13    return binom;
14 }
```

Figure 5.10: An example where either all lines are executed or none. CFI occur rarely.

- Even in the case where multiple executions of a CFI result in different execution flows, the instructions before and after the instructions affected by the CFI may generate redundant coverage data. For example, an if/then/else block may have distinct coverage data for multiple executions, but the code before and after this block is always executed<sup>1</sup>. Conceptually, we remove instructions with different coverage data for multiple executions and track them separately.

<sup>1</sup> It may not happen that these code blocks were not executed, otherwise the if/then/else block would always have the same coverage: none.

Given the diverse list of possible redundancies, we do not identify each instance separately but apply a general approach to detect all redundancies. The coverage data for two lines  $l_1, l_2$  is redundant if the coverage data for  $l_1$  is equal to the coverage data  $l_2$  for all executions of the software, i.e., both lines have the same coverage state in all coverage data of all test executions. In such a case, the state of  $l_1$  determines the state of  $l_2$ . Section 3.3.8 describes the details of the algorithm and shows an implementation.

#### 5.4.2 Evaluation

We investigate the following research question (RQ) in the context of the large-scale software project SAP HANA:

- RQ10 To what degree does each size reduction technique reduce the size of coverage data?
- RQ11 How high is the redundancy of coverage data (in terms of source code lines with equal behavior) for all tests in a large industrial project, and does it change over time?

To answer RQ10, we start with the set of all coverage files that contain line coverage data for all tests within a test run (Section 3.2.5 describes the format). For this initial set of files, we apply the techniques described in Section 5.4.1 in the following order:

1. **Uninteresting data.** For each file, we remove all information about lines that are not executed and further remove all source files that do not contain at least one executed line.
2. **Bitset.** For each file, we use a bitset to store the line numbers, see Section 3.2.5. For practical reasons, we also store the data now in a binary format via Java serialization.
3. **Dictionary compression.** We create a dictionary, which is a bidirectional mapping from filenames to  $\mathbb{N}^+$ . We implement this mapping by a map where the keys of type string are unique and the value for a new key is the current size of the map. Given such a dictionary, we use it to replace all filename strings in all files, i.e., we replace them by the corresponding number. We then store the dictionary in a separate file.
4. **Coverage compaction.** We apply the algorithm described in Section 3.3.8 for all coverage files.

Table 5.4 presents the results for each step. The results show that each step reduces the size of coverage data by 35% to 80%. The actual savings may depend on the order of the techniques. However, as our main interest is the final result, we did not investigate all possible order-combinations.

Step	Size in GB	Comparison in Percent	
		To Base	To Previous Line
Base (initial size)	80.53	100.00	100.00
Keep only executed lines	20.00	24.84	24.84
BitSet utilization	5.10	6.33	25.48
Dictionary compression	3.27	4.06	64.10
Coverage compaction	1.25	1.55	38.12
Baselines			
GZIP level 6 (default)	15.45	19.18	
GZIP level 9	13.95	17.33	
xz/LZMA level 6 (default)	6.11	7.59	

Table 5.4: Results of several size reduction techniques for line coverage data from 2019-12. Techniques applied consecutively from top to bottom.

The exact savings may also depend on the format that is used for the data. For example, we use the Java `HashMap` and Java serialization that both require more space compared to a custom format. Even considering such limitations, we still conclude that each technique provides a valuable size reduction because the effort to implement each of them is rather low. We also consider a final size of less than two gigabytes acceptable for practical use, even if there might be still a wide range of possible compression techniques that could be applied for further savings.

– Answer [RQ10](#) –

Each technique reduces the size of coverage data by 35 % to 80 %. The total size is reduced from 80.53 GB to 1.55 GB, or by 98.45 %.

To answer [RQ11](#), we apply the algorithm described in Section 3.3.8 for coverage data of different coverage runs. A coverage run consists of a set of tests that are executed with coverage measurement enabled. These tests then produce coverage files that we use as an input for our algorithm.

For each coverage run, we calculate the number of executed lines (lines hit) for this run. We then use all coverage data files and apply the algorithm for coverage compaction as described in Section 3.3.8. We then calculate the number of lines hit after the coverage compaction. We call these lines hit *line groups* to distinguish them from the lines hit in the unmodified source. Finally, we calculate the redundancy by dividing the number of line groups by the number of lines hit. For each coverage run, we calculate the redundancy, which is the number of redundant lines over total lines.

Table 5.5 presents the results. The results show that the number of line groups is only 3 % of the number of all executed lines. This ratio is rather stable over time, i.e., within the time frame of one year, the factor between the smallest and largest ratio is nearly 1. We also see that in 2019, although the number of lines hit has increased, the number of line groups grows nearly proportional, therefore resulting in a similar redundancy compared to the year 2016. We assume that this could be correlated to the distribution of x86/x64 instructions. The study of Akshintala et al. shows that several projects have similar distribution for different types of x86/x64 instructions [4]. Based on this empirical evidence, we can assume that the binaries of SAP HANA in 2015 and 2019 also have a similar distribution of

Coverage Run	Lines Hit	Line Groups	Redundancy
2015-11-15	2 901 575	79 741	97.25
2016-05-19	3 172 337	93 162	97.06
2016-08-04	3 371 109	97 368	97.11
2016-10-25	3 510 727	104 764	97.02
2016-10-27	3 501 611	104 355	97.02
2016-10-29	3 422 442	107 402	96.86
2016-11-01	3 421 780	104 837	96.94
2016-11-03	3 399 853	104 638	96.92
2016-11-05	3 424 585	109 338	96.81
2016-11-07	3 413 424	105 235	96.92
2016-11-10	3 405 657	105 361	96.91
2016-11-12	3 391 712	108 754	96.79
2016-11-15	3 436 853	106 030	96.91
2019-09-18	5 111 926	189 863	96.29

Table 5.5: Coverage redundancy for different coverage runs.

instruction types and therefore show a similar ratio of redundancy.

Epitropakis et al. report similar results for smaller software [85]. They report a redundancy of 86.30% to 99.80% for 6 programs with 5 689 to 1 283 504 lines of code. They found the largest redundancy for the largest program. Because our study project is also rather large, we assume that larger programs provide a higher percentage of coverage compaction. We argue that this is reasonably caused by the statistically higher chance of possible redundancies due to the larger amount of lines.

— Answer [RQ11](#) —  
 For SAP HANA, 96.29% to 97.25% of the coverage data is redundant.  
 The ratio seems to be constant over a time frame of 4 years.

### 5.5 Shared Coverage: Test Core Identification

Integration tests typically execute frequently-used program code to perform application startup, tear-down, and a set of core application functions (e.g., parsing of SQL, query execution). Furthermore, both system tests and unit tests call code from many project-internal libraries, e.g., memory management subsystem or string manipulation. This gives rise to the concept of *shared (functionality) coverage*. We define shared coverage informally as line coverage information corresponding to code that is executed by multiple tests. Such code typically includes functionality related to startup/tear-down, common application and utility routines and frequently used data structures for, e.g., memory or string handling. Technically, the shared coverage can be identified as a set of source lines covered by at least  $k$  tests, with parameter  $k$  depending on the project (see Section 5.5.5).

Shared coverage

In general, shared coverage blurs the differences between tests in terms of their coverage, creating several detrimental effects. First, it reduces the specificity of the coverage information, making it more difficult for developers to decide which tests address which parts of code. Analogously, this creates barriers to utilize techniques such as change-centric testing. In the context



of test clustering, shared coverage can pose a challenge for common distance metrics (e.g., Jaccard’s similarity metric [141, 263]) as large shared coverage might dominate more subtle coverage differences. Another problem caused by shared coverage is the increased size of coverage data.

In the context of SAP HANA, we observe a large amount of shared coverage: 20% of the covered lines can be found in 80% of all test suites. Consequently, identification and removal of shared coverage is an important processing step to improve coverage-based characterization of tests.

By *removing shared coverage* and increasing the specificity, we can improve the understanding of the relationship between test and parts of the code that are targeted by the test. Removing shared coverage is highly beneficial for further methods like test prioritization, clustering, or the analysis of test quality. Furthermore, such processing can significantly reduce the size of coverage data. We present approaches for filtering shared coverage in Section 5.5.4, and evaluate them in Section 5.5.5.

Removing shared coverage

Finally, removing shared coverage gives rise to another concept, the *test core*. A test might execute a large amount of code although the purpose of the test is to verify only a specific functionality. For instance, a test executes 500 000 lines of code, but the tested functionality consists of only 50 lines of code. In such a case, we call these 50 lines of code the test core. We further investigate this concept in the following sections.

Test core

### 5.5.1 Examples

We show the practical impact of shared coverage by two types of examples. First, we use a heatmap to visualize the distances between coverage files. Second, we aggregate and analyze the distances analytically.

#### 5.5.1.1 Heatmaps of Distances

We apply the following methodology for the visualization and the analysis:

1. Load a set  $S_{CF}$  of coverage data files.
2. Sort them by a sort criteria  $SC$  descending, i.e., transform  $S_{CF}$  into a sorted list  $C$  where  $e_i \leq e_j \forall e_i, e_j \in \{e_1, \dots, e_n\} = C$  with  $i \leq j$ .
3. Create a matrix  $M$  of size  $|C| \times |C|$  where the rows represent  $C$  and the columns represent  $C$ . The matrix values represent the results of  $d(r, c)$  where  $d$  is a distance function,  $r$  is the current row and  $c$  is the current column, i.e.,  $r, c$  are two coverage files and the value is the distance between these two coverage files.
4. We transform  $M$  into an image where each element  $m_{ij}$  of  $M$  is represented by a pixel with a color. We select the color by a gradient so that  $\min(m_{ij}):color1$  and  $\max(m_{ij}):color2$ . More specifically, based on the distinct values of  $M$ , create a gradient from color  $c_1$  to color  $c_2$  where  $c_1$  represents the smallest value of  $M$  and  $c_2$  represents the largest value. For this purpose, we order the list of distinct values of  $M$  from smallest to largest into the list  $L_M$ . We then associate each item  $i \in L_M$  the (normalized) position  $p_i = \text{index}(i)/|L_M|$  with  $0 \leq p_i \leq 1$ . We use  $p_i$  as a factor for our gradient to calculate the final color. For instance, given  $M$  with element  $m_{ij}$ , we calculate a black-white gradient in a RGB color

system with  $\text{rgb}(p_{m_{ij}} \times 255, p_{m_{ij}} \times 255, p_{m_{ij}} \times 255)$  for each value in  $M$ .

The steps contain several variables that we will explain next. Table 5.6 provides a summary of the following discussion.

- $S_{CF}$ : The set of coverage files. For our examples, we use 2613 coverage files of a coverage run from 2019-09-18, each containing information about 5 500 000 executed lines. We use  $S_a$  to denote the set of all coverage files for the specific test run of 2019-09-18. Additionally, we investigate the subset  $S_s \subset S_a$  that only contains coverage files related to *system tests*.
- $SC$ : We use multiple sort criteria. Namely, we order by name  $SC_{\text{name}}$ , by number of executed lines  $SC_{\text{hit}}$ , by sum of a column  $SC_{\text{columnSum}}$ , and by differences  $SC_{\text{diff}}$ . Note that we apply  $SC_{\text{columnSum}} = SC_{\text{cs}}$  and  $SC_{\text{diff}}$  not on the set of coverage files  $S_{CF}$ , but on the matrix  $M$ , i.e., both criteria operate on columns. In the case of  $SC_{\text{cs}}$ , we calculate the sum for each column and order all columns by their sums. In the case of  $SC_{\text{diff}}$ , starting with the first column  $c_1$  with the smallest sum, we set the next column  $c_{n+1} = \min(|c_n - c_i|, \dots, |c_n - c_j|)$  where  $c_i, \dots, c_j$  represent all columns that are not yet selected.
- $d$ : We use the metrics defined in Section 3.3.3. Namely, the Euclidean metric  $d_e$ , the unshared metric  $d_u$ , the proportional binary metric  $d_p$  and also, although they are not a metric, we investigate the shared function  $f_s$  and the sub function  $f_m$ .

Coverage Files $S_{CF}$	Sort Criteria $SC$	Metrics $d$
$S_a$ All tests	$C_l$ Lines hit	$d_e$ Euclidean
$S_s$ System tests	$C_n$ Name	$d_p$ Proportional Binary
	$C_c$ Column sum	$f_s$ Shared (not a metric)
	$C_d$ Differences	$f_m$ Sub (not a metric)
		$d_u$ Unshared

Table 5.6: Options for several variables.

As we can deduce from Table 5.6, there are 40 combinations. Fig. 5.11 shows a heatmap for each combination. The heatmap pictures have sizes of up to  $2613 \times 2613$  pixels and therefore cannot be presented without loss of details, even considering showing only one image per full single page. Hence, we reduce the sizes of the images even further as shown in Fig. 5.11.

We can see several interesting patterns in Table 5.6. We can recognize multiple black squares within the heatmap of Fig. 5.11.5 (and several other heatmaps). In fact, there are three black squares, due to the scaling the smaller one between the upper-left and bottom-right is not clearly visible. These black squares represent test executions that have a low amount of “unshared” lines, i.e., lines that are only executed in one test, but not the other. These three black squares represent the subgroups of test suites that contain (a) system tests, (b) component tests (not further defined) and (c) unit tests. These patterns are not visible for sorting criteria  $C_n$  as the (start of the) name of a test does not allow to differentiate such tests. Given these insights, we further analyze only system tests, i.e., the group of  $S_s$  (which is at the lower part of Table 5.6 and separated by a horizontal line).

Focusing on system tests ( $S_s$ ), we can recognize a large amount of white values in Fig. 5.11.3. In this case, white means that tests share a large

Black squares represent subgroups of test suites

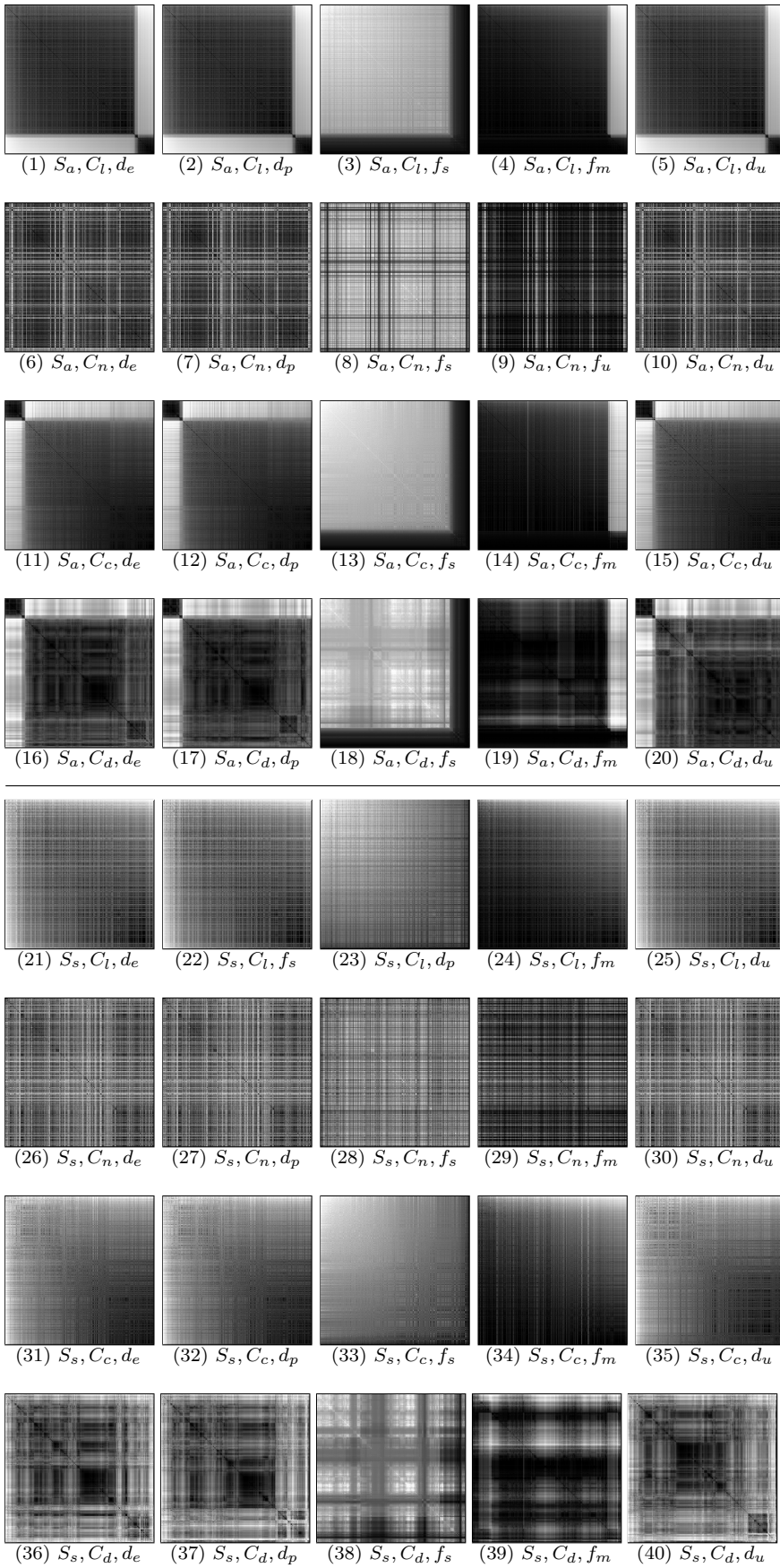


Figure 5.11: Heatmaps representing distances between coverage files. Gradient from black (min) to white (max), calculated for each image separately.

amount of executed lines. This represents the shared coverage as explained in our introduction. The rather large amount confirms our hypothesis that shared coverage occurs frequently and therefore provides us an opportunity for savings. The heatmap Fig. 5.11.38 shows large rectangles of connected white values, showing that  $C_d$  provides rather good results at clustering such areas. However, such clustering has always local optimums and finding a global “best” clustering is rather difficult.

We can also see that heatmaps for  $d_e$  and  $d_p$  are identical. The difference between these two metrics is only a constant factor that does not affect the gradient. The heatmaps for  $d_u$  are similar to the heatmaps of  $d_e$ , but the gradient has a slightly different distribution due to the square root in  $d_e$ . Due to this difference, the sorting criteria  $C_c$  and  $C_d$  result into different heatmaps for  $d_u$  and  $d_e$  as they sort based on the results of the metrics.

Comparing heatmaps for  $C_l$  and  $C_c$  shows two interesting aspects. First, the result for both sorting criteria is rather similar for  $S_s$ , i.e., for system tests. For  $S_a$ , i.e., all tests, the placement of the black squares is on the opposite side in a direct comparison. This is expected due to the implementation of  $C_c$ . We start with the “largest” column. The largest column represents the difference with the most “unshared” lines, i.e., lines that are only executed in one test. We find the largest such differences by comparing the execution of test suites with unit tests to the execution of test suites with system tests. Therefore, such unit test suits will be placed at the top-left for  $C_c$ .

Note that for several heatmaps, we can see a special pattern for the diagonal. This represents the property of metrics, that  $d(x, y) = 0 \Leftrightarrow x = y$ . For  $f_m$ , which is not symmetric, we do not see such pattern and for  $f_s$  we see a white diagonal because the diagonals represent the maximum values when a coverage file shares all content with itself.

In summary, the examples show us that the test executions exhibit a rather large amount of shared coverage and, by comparing coverage files, we can analyze and detect the shared coverage. In fact, the heatmap analysis detected several tests that were nearly identical and some of them were then removed by SAP engineers. Furthermore, we also identified groups of test suites that covered similar functionality (showed as a rectangle of the same color on the diagonal). Such groups were combined into single test suites. Therefore, we conclude that such an analysis can already reduce test costs.

### 5.5.1.2 Analysis of Distances

Our methodology for the analysis contains the following steps:

1. Load a set  $S_{CF}$  of coverage data files.
2. Create the set of pairs  $S_p = \{(x, y) \mid x, y \in S_{CF}, x \neq y, (y, x) \notin S_p\}$ , i.e., all pairs without symmetry (and without self-reflexive pairs).
3. Use  $S_p$  to calculate the set of distances  $S_d = \{d(x, y) \mid (x, y) \in S_p\}$ . Then, we calculate min, arithmetic mean and max for  $S_d$ .

The steps contain two variables:

- $S_{CF}$ : The set of coverage files. For our examples, we use 2613 coverage files of a coverage run from 2019-09-18, each containing information about 5 500 000 executed lines. We use  $S_a$  to denote the set of all coverage files

Large amount of white values that represent shared code

Special pattern for the diagonal

for the specific test run of 2019-09-18. Additionally, we investigate the subset  $S_s \subset S_a$  that only contains coverage files related to *system tests*.

- $d$ : We use the metrics defined in Section 3.3.3. Namely, the Euclidean metric  $d_e$ , the unshared metric  $d_u$ , the proportional binary metric  $d_p$  and also, although not a metric, we investigate the shared function  $f_s$ . We do not investigate the sub function  $f_m$  because  $f_m$  is not symmetric.

Metric	Min	Average (AM)	Max
$S_a$ (all tests)			
$d_e$ Euclidean	9.1	489.8	1 066.5
$d_p$ Proportional Binary	12.9	692.7	1 508.3
$d_u$ Unshared	83.0	279 443.8	1 137 522.0
$f_s$ Shared (not a metric)	1 432.0	555 347.9	1 051 630.0
$S_s$ (only system tests)			
$d_e$ Euclidean	25.1	393.8	784.0
$d_p$ Proportional Binary	35.4	556.9	1 108.7
$d_u$ Unshared	628.0	163 555.9	614 651.0
$f_s$ Shared (not a metric)	503 361.0	696 897.4	1 051 630.0

Table 5.7: Results of distance calculations for coverage files of all test executions.

Table 5.7 presents the results of our analysis. The results show that focusing on system tests indeed increases the shared part as the arithmetic mean (AM) of  $d_u$  is larger for  $S_a$  compared to  $S_s$ . Also, the AM of  $f_s$  is larger for  $S_s$  compared to  $S_a$ . Even more, the min, in this case, is larger by a factor of 350. This means that there is no combination of two system tests that do not share at least 500 000 executed lines.

We also see that min, AM and max of  $d_e$  and  $d_p$  always differ by a factor of  $\sqrt{2}$ . The expected factor by definitions of  $d_e$  and  $d_p$ . The factor propagates to the result due to the distributive property of arithmetic.

In summary, the calculations show that there exists a large amount of shared coverage between all coverage files. Focusing on system tests, this shared part is a significant proportion of the overall execution. Even more, some test executions differ by less than 1% of their execution.

### 5.5.2 Execution Frequency

Software typically has common and specialized functionality. The common functionality is reused several times by different parts of the software. Typical examples for common functionality are input parsing, output formatting or memory and string management in C++ and other languages. In the broader sense, a language standard library also provides common functionality such as data structures like lists or maps. Specialized functionality typically represents a specific requirement or use case for a software project.

In practice, we differentiate *common and specialized functionality* by the frequency of their execution, where the frequency of a line is defined as the number of tests that execute this line. This differentiation is based on the hypothesis that common functionality is called in more tests compared to specialized functionality. This hypothesis is reasonable, as we call functionality common if it is reused several times by different parts of the

Common and specialized functionality

software and therefore it should be used by more tests compared to specialized functionality. Note that we do not define a priori an exact threshold for the frequency separating between common and specialized functionality. Instead, we investigate possible values for this threshold later empirically.

Line coverage data allows us to analyze the execution frequency for each line. More formally, for  $n$  coverage data files  $C = \{c_1, \dots, c_n\}$  representing  $n$  test executions, each line  $l \in \bigcup_{c \in C} c$  has a frequency  $f_l$ :

$$f_l = \frac{|\{c \mid l \in c \text{ and } c \in C\}|}{n} \in \left\{ \frac{i}{n} \mid 0 \leq i \leq n, i \in \mathbb{N}_0 \right\}. \quad (5.3)$$

To simplify the discussion, we may omit the denominator and will only use the (unreduced) execution count in all cases where the denominator does not change. We call this number (i.e., the nominator) the *testcount* as it counts the number of tests that execute a specific line.

Testcount

The following example shows frequencies for several coverage data files:

$$\begin{array}{lcl} C = \{c_1, c_2, c_3, c_4\} & f_1 = |\{c_1, c_2, c_3, c_4\}|/4 & = 4/4 \\ c_1 = \{1, 2, 5\} & f_2 = |\{c_1, c_2, c_3\}|/4 & = 3/4 \\ c_2 = \{1, 2, 4, 5\} & f_3 = |\{\}|/4 & = 0/4 \\ c_3 = \{1, 2, 4, 5, 6\} & f_4 = |\{c_2, c_3\}|/4 & = 2/4 \\ c_4 = \{1, 5\} & f_5 = |\{c_1, c_2, c_3, c_4\}|/4 & = 4/4 \\ & f_6 = |\{c_3\}|/4 & = 1/4 \end{array} \implies$$

In this example, line 6 may represent specialized functionality and lines 1 and 5 may represent common functionality. However, due to the small number of lines, the distinction may be artificial in this specific example.

### 5.5.3 Test Core Identification

Testing a certain functionality within a program  $P$  may require executing other functionality that is not supposed to be tested by the test. For instance, testing a function *isPrime* that checks whether a number given by a single string argument is a prime number will probably also execute a string to integer conversion. On the system test level, a large fraction of a program could be executed although only a small part of it is tested by the test. For the *isPrime* function, a test might only test the prime functionality, but cannot avoid executing the string conversion (given the source code is fixed). We use the term *test core* of a test to indicate the part of the software that is indented to be tested by a test. In some cases, more likely for unit tests, the test core can be identical to the complete execution. In the case of SAP HANA, the difference between the test core and the execution of the test can be several 100 000 lines of code.

Test core

Theoretically, the full coverage data can be separated into shared coverage and the test core. In practice, the distinction between shared coverage and the test core may not be a binary classification due to unclear test intentions, imprecise tests or an insufficient amount of tests to identify the shared part. For instance, the execution of a single test may not allow identifying shared coverage and the test core based on the test execution.

#### 5.5.4 Shared Coverage Removal Approaches

As explained in the previous section, shared coverage, i.e., common functionality, is typically not of our interest. We want to identify the test core, i.e., the specialized functionality that is intended to be tested by a test. Thus, to obtain the test core, we must filter out the shared coverage.

To filter out shared coverage, we have to identify and then remove it. The latter step is a technical detail using the sub operation from Section 3.3.2, therefore we focus on the identification of shared coverage. We investigate three different approaches: differential analysis, baseline removal, and feature extraction.

##### 5.5.4.1 Differential Analysis

Our main idea is based on the fact that shared coverage lines are hit by more tests than other lines. We can thus proceed in two steps:

1. Compute the testcount for each line  $l$  (the number of tests executing  $l$ ).
2. Mark (or immediately remove from coverage data) all lines with a higher testcount than a given threshold.

For small threshold values, e.g., a threshold of 1, the algorithm removes all shared coverage lines. For high threshold values, this will remove only lines that are covered by nearly all tests. Both extremes are not optimal.

An example of removing too much are two tests that cover two branches of an if statement in a function. We cannot prevent that both tests cover the condition of the if statement. However, a removal is probably not desired because the condition is part of the if statement.

On the other hand, if the threshold value is too high, we do not remove anything. We could set the threshold to the number of tests (minus 1) to filter all common functionality. However, a test for an exception could abort the execution before reaching such common functionality. In such a case, although expected, the common functionality would not be filtered.

Therefore, we must identify a reasonable threshold. We find this threshold by analyzing the distribution of the different testcount values versus the number of lines with the given testcount. We then either manually conclude a threshold based on a visual analysis of the graph, or we implement an algorithmic approach that automatically detects changes in the slope within a sequence of such distribution data.

Threshold identification

##### 5.5.4.2 Baseline Removal

Our main idea in this variant is the assumption that shared coverage occurs because part of the code has to be executed for all or most inputs. This gives rise to creating a test that executes only such code.

We implemented it as a test with only the most fundamental functionality. In our case (SAP HANA is a database system) this is a test with only simple `create`, `insert`, `select`, `delete`, and `drop` statements. The coverage collected for this test is the baseline and can be removed from all other coverage files. As an alternative for creating the baseline test manually, it is also possible to find a suitable candidate automatically with a combination of the previous approach based on testcounts.

### 5.5.4.3 Feature Extraction

A typical software project has a hierarchical directory structure based on conventions on how to organize related files. Therefore, each directory may represent a set of features. In our case, the common and auxiliary functionality (e.g., memory or string handling) have own directories. Therefore, we can remove all coverage for files in these directories. We do not provide an evaluation for this approach, because the effect is obvious and the removal of a predefined set of directories only solves the problem partially.

### 5.5.5 Evaluation

We investigate the following research question (RQ):

RQ12 How does the removal of shared coverage improve the specificity of code coverage data in the context of SAP HANA?

We answer RQ12 in 3 steps:

1. We identify shared coverage with the following two approaches from Section 5.5.4: **differential analysis**, and **baseline removal**.
2. We remove the shared coverage.
3. We analyze test specificity before and after the removal of shared coverage.

**Step 1:** For the approach based on **differential analysis**, we need a threshold for removing lines with a high testcount. To find this threshold, we analyze the distribution for testcounts per source code lines, see Fig. 5.12 for the data of 2017-01. We select the threshold  $DB_{80}$  at 80% which corresponds to 238 test suites, i.e., every line hit by more than 238 test suites is removed. We select two additional thresholds for our evaluation below and above  $DB_{80}$ :  $DA_{60}$  and  $DC_{90}$ . To automate the threshold selection, it is possible to utilize the slope of the distribution curve.

Fig. 5.13 shows the distribution plot for the data of 2019-11. The shape of the distribution plot is similar to Fig. 5.12 and remains over time.

For the **baseline removal** approach, we manually chose the baseline test suite in consultation with SAP engineers.

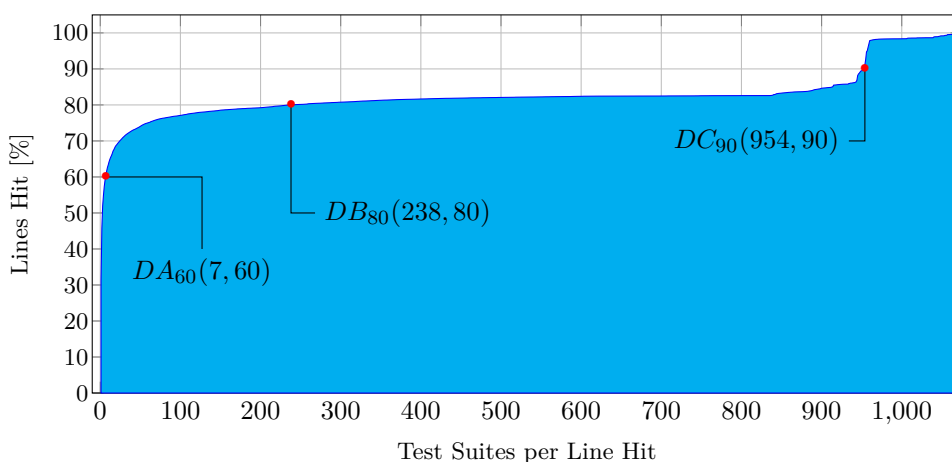


Figure 5.12: Distribution plot for test-count versus the relative amount of covered lines. Data of 2017-01. E.g., 80% of all lines hit are covered by  $\leq 238$  test suites and 31% of all lines hit are covered by  $\leq 1$  test suite.



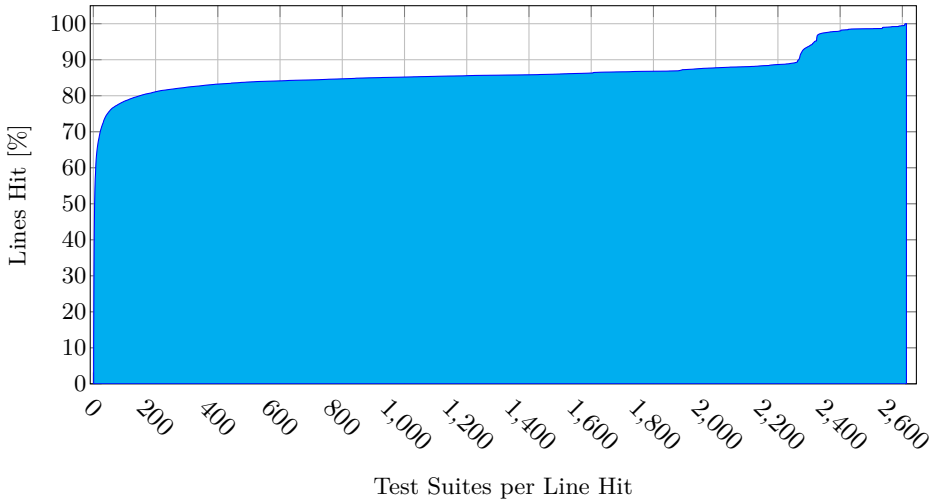


Figure 5.13: Distribution plot for test-count versus the relative amount of covered lines with at most testcount value of tests based on data of 2019-11.

**Step 2:** After the identification step, the removal step uses the *sub* operation (See Section 3.3.2), either with the baseline test suite or with an artificial coverage file made of all lines with a testcount above the current threshold. Table 5.8 shows the size reduction after the *sub* operation. The baseline removal reduced the coverage size to 36.01 % of the original coverage.

Threshold [%]	Testcount	Size Left [%]
0	0	0.00
30	0	0.00
60	7	1.93
70	25	4.64
80	238	15.06
90	954	58.44
99	1 046	95.13
100	1 065	100.00

Table 5.8: Relative size of coverage data (2017-1) after removal of lines with certain testcounts. E.g., lines hit by  $\leq 238$  test suites represent 80 % of all lines. Removal of lines with testcount  $> 238$  reduces the data to 15.06 % of the original size.

**Step 3:** After the removal, we measure the test specificity before and after applying our approaches. We evaluate the specificity by counting the number of lines hit in source code located in different (component-specific) subdirectories and asking SAP engineers to verify whether the directory relates to the test intent or not.

We select 20 test suites randomly and compute the top five coverage directories for each test suite. These are the directories with the highest number of lines hit for all files in the directory. Under the assumption that the directory layout represents a coupling between source files and modules, we should see an increase in specificity for the top directories.

We represent the 20 test suites and five directories for each test suite by a  $20 \times 5$  matrix. We create this matrix for six configurations. For each threshold  $DA_{60}$ ,  $DB_{80}$ ,  $DC_{90}$  (see Fig. 5.12), and for each version with the original data and after the removal of lines with high testcount.

We showed these six matrices to developers of SAP and asked them to select 10 test suites where they can judge which directories are specific to this test. They should highlight all specific directories within the top 5 directories in green, the unspecific directories in red and the rest in yellow.

After this manual task, we attribute the following scores to the classifications. A green field gets +1, a red field -1, and a yellow field gets 0. In addition, we add a top score for the first  $n$  correct top directories: If the top  $n$  directories are green, we add additional  $n$  points; for red, we substitute  $n$  points. For instance: (g,g,r,y,y) gets +2 for green, -1 for red, 0 for yellow and +2 for top-2 green, in total:  $2 - 1 + 0 + 2 = 3$  points; (y,r,r,r,g) gets  $(1 - 3) + 0 + 0 = -2$  points.

	Case	Correct Dir.	Score	Lines Hit
Differential analysis				
$DA_{60}$	before	1/10 (7 wrong)	$-28-25=-53$	2 737 700
	after	8/10 (0 wrong)	$5+10=15$	37 848
$DB_{80}$	before	3/10 (7 wrong)	$-35-26=-61$	3 811 208
	after	8/10 (1 wrong)	$-16+9=-7$	292 087
$DC_{90}$	before	0 (10 wrong)	$-46-46=-92$	3 038 125
	after	1/10 (9 wrong)	$-30-20=-50$	1 178 414
Baseline removal	4/10 (6 wrong)		$-24-6=-30$	728 794/3 811 208

Table 5.9: Evaluation of test specificity changes for shared coverage removal. The score depends on manual evaluation (see Section 5.5.5), higher is better. Correct dir. indicates the amount of correctly identified directories.

Table 5.9 shows the results of our evaluation. The shared coverage removal based on differential analysis improves the specificity of the coverage. The highest increase in test specificity was possible with the highest testcount removal. This is expected since the highest specificity would be reached if we only keep lines that are hit by exactly one test. But as discussed in Section 5.5.4, this is not desired. This choice would remove all covered lines with multiple tests, but some of them are still interesting for further analysis. In summary, the 80% threshold provides a good increase in test specificity but still guarantees multiple test coverage for most code parts.

We evaluate the baseline removal in the same way. We select a baseline test that covers only the basic functionality `create table`, `insert`, `select`, `delete`, `drop table`. The score for the baseline removal is -30. Therefore, the test specificity is lower compared to the results for  $DB_{60}$  and  $DB_{80}$ .

#### Answer RQ12

Both approaches, differential analysis, and baseline removal, improve the specificity of code coverage and allow identification of the test core. In a direct comparison, differential analysis correctly identifies a larger amount of source code as a test core compared to the baseline approach.

## 5.6 Nondeterminism in Testing

### 5.6.1 Random Coverage

We understand by *random coverage* the phenomenon that a line of code is sometimes executed, and sometimes not executed (and thus covered or not) in multiple executions of the same test with the same state of the software. Thus, we see randomness at the level of coverage data. Other work already observed the effect of random coverage [186]. Apart from obtaining less reliable data, random coverage creates issues for test similarity metrics, and in general for all techniques that depend on accurate coverage data.

Previous work already mentions several sources of random coverage [186]. For SAP HANA, we have detected the following sources of random coverage:

- functions providing random numbers,
- time/date creation,
- multi-threaded execution,
- memory handling,
- scheduling,
- file system interactions,
- errors of coverage tools.

For some test suites of SAP HANA, we found differences of covered lines for two identical coverage runs from 50 to several hundred lines. This is only a small fraction of the total number of lines covered by a test suite which typically ranges (after shared coverage removal) from 10 000 to 200 000. However, if line coverage of two such test suites *differ* by, e.g., only 100 LOC, the random coverage becomes critical, as we do not know which fraction of this difference can be attributed to randomness. Fig. 5.14 shows that the same test can generate different coverage for multiple executions. These different coverage data appear to have no relation, i.e., the underlying causes of the differences appear to be random.

	A	B	C	D		A	B	C	D
A	0	2 266	2 579	2 963	A	0	138	158	182
B	426	0	747	967	B	75	0	95	100
C	476	484	0	801	C	74	75	0	83
D	1 205	1 049	1 146	0	D	115	103	101	0

(a) Number of executed lines. Values calculated by row-column.

(b) Number of source files left. Values calculated by row-column.

Figure 5.14: Effects of random coverage. Line coverage data A,B,C,D are the result of repeated executions of the same test. The test executes about 40 000 source code files and 4 000 000 lines of code. For Tables (a) and (b), the entries represent results of (row-column), i.e., row 2, column 1 shows the results for B-A. Operations as defined in Section 3.3.2.

Operation	#Executed Lines	#Source Files Left
$A + B + C + D - (A \& B \& C \& D)$	4 777	316
$A - (B + C + D)$	2 101	106
$B - (A + C + D)$	125	28
$C - (A + B + D)$	262	35
$D - (A + B + C)$	869	70
$B + C + D - A$	1 642	166
$A + C + D - B$	3 482	222
$A + B + D - C$	3 745	243
$A + B + C - D$	3 400	231

(c) Results for several operations.

### 5.6.1.1 Approach

Randomness in coverage data is a threat to the soundness of any further analysis on coverage data. For instance, approaches that assume exact coverage for change based testing may produce different results depending on the effects of the observed randomness. The best approach to remove

random coverage would be to remove all sources of randomness. However, this is typically not feasible in a large project for multiple reasons:

- Identifying all sources of randomness and replacing them with a deterministic behavior is difficult and rather time consuming.
- Introducing implementations that are only used for testing creates a special test-state of the software. Test executions would therefore not test the behavior of a productive environment.
- Removing all sources of randomness would require disabling concurrent execution. This would not only increase the time for test execution tremendously but also prohibit any testing of concurrent behavior.

Given these points, the removal of all sources of randomness is not a feasible approach. Therefore, we consider the following strategies:

1. We design any of our approaches that utilize coverage data in such a way that random coverage is either not a threat or we control the effect.
2. We detect all lines of source code that show random behavior and mark them for deletion or for future analysis.
3. We rerun a test multiple times to harmonize the different sets of lines hit created by randomness. We can either create the union of coverage data for all runs (which contains all randomly covered lines) or create the intersection for all runs (which does not contain any randomly covered lines). Manual evaluation of this approach for single test suites showed reasonable results. However, we skipped an evaluation on a larger scale due to the constraints on execution time and resource costs.

Strategy 3 increases the time for test executions by the rerun factor. We may have to run all tests multiple times (with coverage-enabled overhead). Therefore, in practice, we apply it only for a small, selected subset of tests.

Regarding strategy 1, the approaches presented in Section 5.5 for shared coverage removal control the effect of random coverage by design. As randomness may be encountered by all tests, each test may have a percentage to execute a specific line. For some lines, this percentage is 100 %, for other lines, it is lower. Thus, statistically, the randomness modifies the number of tests that execute a line (the testcount) by the percentage factor. As a result, shared coverage removal typically also removes random coverage.

To handle edge cases of random coverage, such as a single line that is only executed by a specific single test in only 5 % of all executions, we can use the strategy 2. We can interpret multiple coverage runs as a set of time-series over source code lines, i.e., we create for each line a binary sequence of ones and zeros that represent, over time, whether this line was executed or not. We can create these time-series either per test run or per test. Per test run may allow fast detection of lines that show a low probability of execution. Per test allows identifying randomly executed lines for each test. Although we expect this approach to be promising, we did not evaluate it due to practical issues. Tracking code (and test executions) over time in a large project is a rather complex tasks as shown by Sections 4.2 and 4.3.

For the rest of this work, if not specified otherwise, we assume that the effects of random coverage are mitigated by shared coverage removal. To our experience, this assumption is typically true in practice.

### 5.6.2 Flaky Tests

We call a test flaky if the test shows different results (pass/fail) in multiple runs under the same conditions (inputs, local environment, software version) [181]. The reasons for such behavior are diverse and include issues with the test environment (e.g., file servers), performance impact of other applications, “junk” data created by previous tests, timing/async issues, resource leaks, and randomness due to concurrency issues in the application or the operating system [169, 181].

Empirical data from Google indicates that up to 16% of all tests at Google show flaky behavior [193]. Microsoft reports that 14% to 52% of all builds show flaky test failures [169]. A study on 61 Java projects found that “18% of test suite executions fail and that 13% of these failures are flaky” [168]. Similarly, the Firefox testing process also contains a considerable amount of flaky tests [215]. This number is similar for SAP HANA, where we also find a non-negligible number of flaky test results.

10% to 20% of all tests show flakiness

Flaky tests create a threat to the validity of results for approaches exploiting historical test results, and to a certain degree also for research based on coverage data. A large amount of previous work in this domain assume perfectly stable test conditions, and repeatable, deterministic test results. This might not be the case for projects above a certain size, which calls for an evaluation of such approaches for very large projects.

To eliminate the impact of flaky tests on results in this work, we run a test up to four times if a previous run fails, and we keep the coverage of failed test runs. There are several projects in the context of SAP HANA that attempt to classify test results as correct or erroneous utilizing techniques from machine learning (specifically, scalable SVMs).

## 5.7 Threats to Validity

We discuss possible threats to the validity of our work.

**Test Suites Granularity:** Our coverage data is based on test suites that include a set of tests. However, in practice, even single test cases call complex logic and include several checks.

**Flaky Tests:** As explained in Section 5.6.2, flaky tests influence all analyses based on test success. Therefore, we avoid any approaches that rely on the correctness of a test result.

**Safeness of TCS:** Our TCS strategies are not safe. They can omit test cases that would otherwise have detected faults. This must be considered during a risk analysis of the costs and benefits as we show in Section 5.2.

**Random Coverage:** We observed random coverage. The amount is rather low compared to the total coverage size. Hence, the impact on our results is marginal. In practice, shared coverage removal typically removes random coverage and further approaches were not required.

**Relation between Coverage and Bug-Finding Ability:** Some approaches may assume that the bug-finding ability of a test suite  $TS_1$  and a  $TS_2$  is similar, if  $\text{coverage}(TS_1) = \text{coverage}(TS_2)$  (without further defining the similarity here). This seems to be an undecided research question, see Chapter 4 for further details and discussions.

**A Database Is a Special Environment:** By design, every call of a database executes a large shared part of the database stack. We argue that this shared design applies in fact to a wide field of software branches. Software following the MVC pattern, such as any software with a GUI, has a large shared part. Software with a single entry point is another common example, such as parsers, I/O software or command line tools.

**Tests Are Not Independent** The test independence assumption does not always hold [267]. We also observed this for SAP HANA. This does not affect our work, because we use coverage on a test suite level. Test suites are independent because each test suite is executed completely separated.

**Definition of Line Coverage:**

We defined line coverage in Section 3.1.5. This definition also defines a “line” as a sequence of characters followed by a line break. However, the position of a line break is decided by a developer and can therefore be at arbitrary places. This can lead to several effects:

- All source code is written in a single line without line breaks.
- The amount of line breaks depends on the developer.
- The position of line breaks depends on the developer.

In the first case, line coverage will be meaningless. In practice, this extreme case does not occur because the resulting source code is considered unreadable for humans and not maintainable.

In the other cases, if the amount and positions depend on the developer, a style guide may formally decide the placement of such line breaks [152] and therefore reduce the ambiguity. Such a style guide can be automatically enforced by tools. Even then, developers can choose different implementations for the same task which results in a varying amount of lines.

We conclude that it may not be possible to strictly control this threat. However, this threat may not affect results in a large corpus because we do not expect large deviations. In our work, we typically compare line numbers only by their magnitude. For instance, if a program  $P_A$  contains 212 374 lines of code (LOC) and a second project  $P_B$  contains 5 312 413 LOC, then we conclude that  $P_B$  is “larger” by a magnitude. However, a third project  $P_C$  with 5 313 880 LOC would be of the same size as  $P_B$ . The difference of fewer than 5 000 lines is not significant.

## 5.8 Conclusions

We described and discussed several techniques for test cost reduction in large projects. Our presentation shows that large projects exhibit several specific characteristics that are often not considered in related work for test cost reduction. Conclusively, it is important to design approaches for test cost reduction in large projects that are tailored to the specific characteristics of large projects. This motivates our main contribution, dynamic unit test extraction in Chapter 6, which is focused on reducing the negative effects generated by system tests with large execution times.

We also presented several approaches that target the characteristics of large projects, such as multiple testing stages or test core identification. We use these approaches as building blocks for our dynamic unit test extraction.

# 6 | Dynamic Unit Test Extraction

In this section, we introduce our main contribution, a practical approach to reduce test costs by reducing the time spent on executing tests without negative effects on the overall quality of the software. The approach is practical because it is implemented for a large real-world software application and it is designed in such a way to fulfill several requirements of practitioners regarding the maintainability of tests and overhead for practical usage.

As we discussed in Chapter 1, tests with large execution times, say 30 minutes or more, contribute substantially to the test costs of large projects. Therefore, we aim to reduce costs created by such tests. The core technology to tackle this problem is *dynamic unit test extraction*, i.e., we analyze the execution of tests with large execution times and use this information to extract one or multiple smaller tests.

This technique combines several aspects of this work. We use static analysis provided by our own Clang compiler plugin (Section 6.1.4.2 and Section 6.2.5) and test core detection (Section 5.5) based on coverage data (Chapter 3) to identify the part of a test execution that is important to extract. We then monitor the execution of a test and use time-travel debugging (Section 6.1) and object creation techniques (Section 6.2) to dynamically extract source code for C++ unit tests. We optionally reduce the number of unit tests by coverage-based test suite reduction (Section 3.3.4). Finally, we use a multi-stage testing strategy (Section 5.2) to efficiently reduce test costs while maintaining the same level of quality for the software under test.

Previous work on extracting behavior from system tests use terms such as test extraction/factoring [123, 224, 225], carving [80, 81], capture and replay [142, 146, 207], or codelet extraction [42].

A large fraction of related work proposes to store the recorded information in a binary format [80, 81, 123, 146]. This has several practical drawbacks. A binary format is neither human-readable nor maintainable, resulting in tests with very limited value in practice according to the feedback of our industry partner. Developers strongly advocate having tests that can be understood and maintained by humans.

Our work is, to the best of our knowledge, the first effort to investigate dynamic unit test extraction for C++ . We believe that C++ proposes several challenges to be handled separately compared to other languages such as Java, Python or R. Related work investigated test extraction for R [167], Java [146, 224], LLVM-IR [42], or FORTRAN [123, 157]

We provide further details about related work within the corresponding sections for our approaches, see Sections 6.1.8 and 6.2.8.

Dynamic unit test extraction

Disadvantages of binary formats

Different approaches for various programming languages.

We focus in this chapter on our approach for dynamic unit tests extraction and its requirements. We first introduce our approach based on time-travel debugging and then describe a technique that finds options to create objects. Finally, we also discuss an approach for automated mock proposal.

## 6.1 Dynamic Unit Test Extraction via Time-Travel Debugging

Compared to system tests, unit tests execute faster and allow more precise fault localization. Consequently, it can be beneficial to replace a system test by an equivalent suite of unit tests. However, writing unit tests for complex applications can require more development effort. Hence, techniques have been proposed to extract unit tests from system tests. However, these approaches have limitations such as serialization of object states that causes hard-to-maintain and partially incomprehensible test code, scalability issues due to a whole project approach, or missing support for C++ .

We address these issues by an adaptive approach for extracting *code-only unit tests*. As a central technical element we exploit *time-travel debugging* for efficient and accurate reconstruction of object states. The extracted unit tests mimic relevant parts of the system tests, and, by displacing the latter in early testing stages, save resources and facilitate fault localization.

Our evaluation on SQLite and SAP HANA indicates that there is a large potential for test cost reduction with our technique. The extraction process overhead is feasible, with an average slowdown factor of 14 for SAP HANA. The acceptance of generated tests is high for SAP HANA, as all of 789 extracted unit tests have passed code reviews and were accepted by developers. Finally, compared to automatic unit test generation via symbolic execution, our technique can achieve higher line coverage in specific cases.

### 6.1.1 Introduction

We focus on reducing the costs associated with regression tests. Such tests are re-run after essential code changes to ensure that previously developed and tested software keeps its behavior. Within the workflow of continuous integration, regression tests are executed for every change that is merged to the main code line which can result in frequent executions. This is still acceptable for fast running unit tests, yet for long-running system tests, a high execution frequency can be prohibitive in terms of waiting time and hardware usage. In addition, system tests typically make fault localization more difficult and are more likely to exhibit non-deterministic behavior [181]. Table 6.1 summarizes our observed differences. Thus, if a system test focuses on a particular functionality or a code region, a suite of equivalent unit tests might be a better choice instead.

	System Test	Unit Test
Coding effort	low	high
Execution time	medium to very large	short
Fault localization	difficult	precise
Test flakiness	frequent	rare

Code-only unit tests  
Time-travel debugging

Disadvantages of tests with large execution times.

Table 6.1: Contrasting unit tests and system tests based on the observations for SAP HANA.



In complex software projects, the effort of coding a system test (in particular when targeting a small part of the code) can be much lower than writing a suite of equivalent unit tests. We observed this phenomenon in the development process of SAP HANA. In this project, system tests leverage a Python framework and typically consist of a simple SQL statement that is executed and the result is checked for correctness. Such new system tests are simple to write and to maintain. In contrast, coding a single unit test (which is written in C++) may require multiple hours. This can be attributed to several factors: a complex process of setting up the compilation targets, manually writing unit tests in a test framework (involving preprocessor macros), identification of object creation steps, and, for legacy code, modifications to a complex C++ codebase to make it testable.

Creating unit tests can require more effort than creating system tests.

Consequently, developers are more inclined to write system tests even if they want to test only a specific functionality of the product. However, the resource demand incurred by the system tests is substantial. Therefore, we propose *dynamic unit test extraction*, or *test carving* [81, 167] to reduce the resource consumption. To this end, a system test is executed in an instrumented environment, and values of input arguments and outputs for targeted functions are recorded. Then, we use this data and a template framework to generate the source code of the unit tests.

Dynamic unit test extraction

Prior work on carving targeted Java [81], FORTRAN [123], or R [167]. However, these techniques have limitations for large projects such as:

1. object serialization in a custom format,
2. missing support for C++ ,
3. scalability issues for large projects with millions of lines and functions.

We address these issues by proposing a scalable approach for extracting “source-code-only” unit tests in C++ . Our method utilizes *time-travel debugging* [84, 250], i.e., reverse-execution of a program flow to reconstruct object creation and modification. Therefore, only source code is generated, and any “non-code” representation (e.g., binary data, XML) is not required for the unit test setup. This improves the maintainability of the extracted tests ([220]) and increases their acceptance by developers at SAP HANA.

Time-travel debugging

Moreover, we specifically target C++ . Several approaches for Java [81, 224] require substantial modifications to work in C++ regarding the memory model and availability of runtime information about internal program state. Moreover, authors of previous work for FORTRAN and R question whether their approaches can be transferred to C++ [123, 167]. Finally, we use differential analysis on coverage data to identify test intentions and reduce the number of generated unit tests. Our contributions are:

- An approach based on time-travel debugging for generating maintainable C++ code for unit test setup.
- A prototypical implementation for extracting maintainable unit tests in C++ source code by capturing data from executions of system tests.
- A process for reducing test costs by combining a differential analysis on coverage data, unit test extraction, and a multi-stage testing strategy.
- An evaluation of our approach in terms of potential test time savings, extraction overhead, developer acceptance, and coverage comparison against test generation via symbolic execution.

### 6.1.2 Motivation

In this section, we illustrate via examples how time-travel debugging is leveraged to extract maintainable unit tests. For confidentiality reasons, we cannot show source code of SAP HANA. Therefore, we use examples from SQLite [121]. SQLite provides 1000 `<name>.test` files implementing 40 000 test cases for the publicly available test suite (so called TCL Tests) [121]. For instance, `main.test` tests `main.c` and contains tests such as:

```
1 do_test main-1.8 {
2   db complete {DROP TABLE "xyz";}
3 } {0}
```

An analysis reveals that a large amount of test cases in `main.test` test the function `int sqlite3_complete(const char *zSql)` in `complete.c`, a function with 158 lines and 41 executable lines of code (Section 6.1.3.2). While executing `main.test`, we record the values of arguments and return values for `sqlite3_complete`, and generate 52 unit tests with distinct sets of arguments for this function. These test cases can be reduced to only 4 tests with identical cumulative line coverage (Section 6.1.3.5):

```
1 TEST(SimpleExample, test1) { //default template
2   int expected = 0; // extracted oracle
3   int result = sqlite3_complete("DROP TABLE \"xyz'\");
4   EXPECT_EQ(expected, result);
5 }
6 TEST(SimpleExample, test2) { // short version
7   EXPECT_EQ(0, sqlite3_complete("\n /* */ EXPLAIN -- A comment
8     \n CREATE/**/TRIGGER ezxyz12 AFTER DELETE backend BEGIN\
9     n UPDATE pqr SET a=5;\n"));
10 }
11 TEST(SimpleExample, test3) {
12   EXPECT_EQ(0, sqlite3_complete("\n CREATE TRIGGER xyz AFTER
13     DELETE [;end;] BEGIN\n UPDATE pqr;\n"));
14 }
15 TEST(SimpleExample, test4) {
16   EXPECT_EQ(1, sqlite3_complete("\n CREATE -- a comment\n
17     TRIGGERX tangentxx AFTER DELETE backend BEGIN\n UPDATE
18     pqr SET a=5;\n"));
19 } // we omit repeated whitespaces for brevity
```

These 4 tests execute 10 times faster than `main.test` without reducing line coverage. As one of the main purposes of `main.test` is testing the `complete` function, we could probably run the extracted tests instead of the large test suite (Section 6.1.3.6). This example shows the overall process and illustrates potential test costs savings.

Since the function is fairly simple and the tests in `main.test` are probably already designed as unit tests, we also provide a more complex example that involves object creation. We identify a central function of SQLite in `hash.c` and trace its arguments in order to extract a unit test:

```
1 *sqlite3HashInsert(Hash *pH, const char *pKey, void *data)
2 // example data when we execute table.test:
3 pH: 0x16327e8->{htsize=0, count=1, first=0x15bcd10, ht=0x0}
4 pKey: 0x153f678 -> "NOCASE"
5 data: 0x153f600 -> ?
```

The function requires object `pH` of type `Hash`, a pointer to a string, and an argument `data` of an void type pointer. For a developer who aims to create an unit test, creating code for correct instantiation and initialization for the arguments `pH` and `data` might require a lot of effort. By leveraging

a time-travel debugger (active during execution of a system test), we can “execute backward in time” to reveal the construction of object `pH`:

Reverse execution

```

1 -> reverse execute
2 Hardware access watchpoint 6: *0x16327e8
3 Value = 0
4 sqlite3HashInit (pNew=pNew@entry=0x16327e8)
5 {htsize = 0, count = 0, first = 0x0, ht = 0x0}
6
7 -> reverse execute
8 Hardware access watchpoint 6: *0x16327e8
9 Value = 0
10 0x00007fd19f2eaa2b in __memset_sse2 () from ...
    
```

Based on this data, we can construct `pH` by allocating memory for it and subsequently calling `sqlite3HashInit`. To finalize the initialization of `pH`, we must set `count` to 1 and construct the object referenced by `first`. To create the third argument `data`, we perform another reverse execution:

```

1 -> reverse execute
2 Hardware access watchpoint 7: *0x1632890
3
4 Old value = 23275784
5 New value = 0 // rev. exec. => old/new switched
6 findCollSeqEntry (create=1, zName=0x52e828 "NOCASE", db=0
7   x16325e0) at ...
8 165 pColl[0].zName = (char*)&pColl[3];
    
```

The construction of `data` requires object `db`, which is a central `sqlite3` data structure, that can be created by the same recursive process.

The extracted unit tests could replace a system test as in `table.test`, resulting in a time reduction by a factor of 100. In the case of SAP HANA, the execution times of system tests vary between 30 minutes to several hours. Replacing them by unit tests can result in a speedup factor of up to 10000. Therefore, extracting unit tests from system tests is beneficial and time-travel debugging allows us to extract maintainable unit tests.

### 6.1.3 Approach

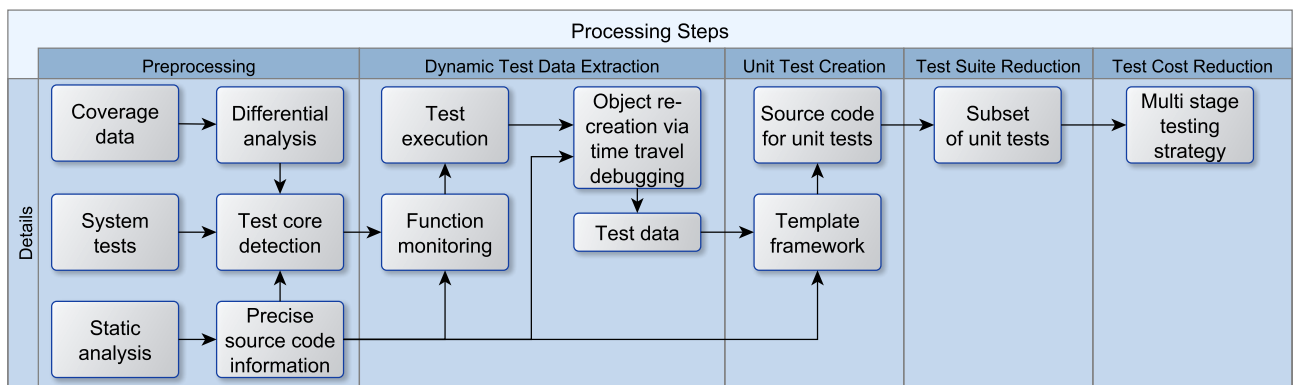


Figure 6.1: Overview of our approach for test cost reduction via dynamic unit test extraction.

The examples in Section 6.1.2 show that several challenges must be solved to dynamically extract unit tests from system tests if the unit tests should be maintainable and the collection process should be feasible for large projects. This section explains the details of our approach to solve these challenges. Fig. 6.1 presents an overview of the individual steps of our approach.

### 6.1.3.1 Definitions

We use the definition of (line) coverage as provided in Chapter 3. We use the term *unit test* to describe a test that tests only a small unit of the code, and the term *system test* to describe a test that executes the complete system. Note that in practice developers do not always adhere to these definitions.

### 6.1.3.2 Preprocessing

In this step, we gather information about the C++ source code.

**Static Analysis:** We gather accurate information about the C++ source code, notably a list of all types, functions, and arguments. Later steps require such information as a binary-only analysis would not be sufficient to create the C++ source code for the extracted unit tests.

**Test Core Detection:** We define the *core of a test* to be the part of the source code that is intended to be tested by the test. We focus on a function level, and so our test core is always a set of functions  $F$ . For unit tests, the test core typically contains exactly one function. For system tests, the test core can contain a large number of functions.

We identify the test core by analyzing the coverage data as explained in Section 6.1.4.2. Alternatively, a developer can manually specify a function to investigate to reduce time for manual investigation and test code writing.

For SAP HANA, system tests typically execute more than 700 000 lines of code. However, we noted that frequently, the purpose of a system test is not to test all of these lines, but to test for some specific functionality that comprises only a small subset of all lines, the test core. A calculation of test similarities (Table 6.2) supports this argument.

System tests execute over 700000 LoC.

### 6.1.3.3 Dynamic Test Data Extraction

In this step, we execute the set of functions  $F$  identified by the previous step to dynamically extract all required data for the generation of unit tests.

Conceptually, we want to log arguments and return values for each call of a targeted function  $f$ . Section 6.1.4.3 provides a brief discussion of multiple approaches we investigated and partially adapted. In practice, our implementation is adaptive to the types of arguments for  $f$ , i.e., a function with only arguments of type fundamental ([138], 6.9) requires less effort than a function with complex objects as arguments. In this section, we focus only on the most advanced approach for complex objects.

We use *time-travel debugging* [22, 23, 84, 155, 205, 206, 250, 253] to follow the creation and modification of complex objects backward in execution time. Time-travel debugging allows us to interrupt the program execution at the entry of a specific function and for each function argument of type object, we can follow the execution flow of the program in a reverse manner and *watch* the object modification and creation. This approach provides an accurate but also fast way to observe all required steps for the object setup.

Time-travel debugging

To execute the functions in  $F$ , we execute the original system test, or, based on coverage data, all tests that execute a function in  $F$ . During the test execution, we record for each function in  $F$  the argument and return values, and, if required, more data such as the corresponding objects.

#### 6.1.3.4 Unit Test Creation

In this step, we use the results of the dynamic test data extraction and a template framework to generate the source code for the unit tests. We use the observed argument values to execute the function in a unit test and we use the observed return value (and more data depending on the specific function details) as a test oracle. We assume that the system test success implies the successful execution of all intermediate functions.

#### 6.1.3.5 Test Suite Reduction

In this optional step, we reduce the amount of extracted unit tests. We apply coverage-based reduction as explained in Section 6.1.4.5. Our work does not focus on this optional aspect since we received mixed feedback from developers of SAP HANA about the benefit of this step. One group of developers suggested keeping all tests because the generation is automated and the execution time is very low. Another group suggested keeping the number of unit tests “reasonably” low to support further maintenance.

#### 6.1.3.6 Test Cost Reduction

In this step, we reduce the test costs by replacing the original system tests with the extracted unit tests for the test core. We assume here that the extracted unit tests evaluate the test core similar to the system test.

Under this assumption, the unit tests can replace the system test or, more realistically, the system test can be executed less frequently, i.e., moved to a later testing stage. A testing strategy that involves multiple stages provides a solution to reduce test costs in a safe way, i.e., the costs of frequent system test runs are reduced by the dynamically extracted unit tests and infrequent system test runs prevent any loss of quality for the final project. See Section 5.2 for further details about the multi-stage testing strategy.

Unit tests displace system tests.

### 6.1.4 Implementation

We highlight in this section several key aspects of our implementation.

#### 6.1.4.1 Static Analysis

We implemented a Clang [171] plugin for performing the static analysis. For the complete C++ source code of SAP HANA with several million lines of code, we gather 1.60 TiB of information about:

- Functions ( $\approx 3.20$  million), e.g., source file, line numbers, signature, return type, namespace, visibility
- Types ( $\approx 600\,000$ ), e.g., source file, line numbers, inheritance, namespace, constructors, member functions, friends, members
- Arguments of functions ( $\approx 3$  million), e.g., name, type, modifiers

As Clang successfully compiles the source code of SAP HANA, we assume that information provided by the Clang plugin is accurate and complete. For instance, Clang resolves type aliasing, handles preprocessor macros transparently, supports templates, informs about inlined functions and implicitly generated functionality, or recognizes class hierarchies and object members.

A Clang compiler plugin provides accurate information.

### 6.1.4.2 Test Core Detection

We collect coverage data for all system tests as described in Section 6.1.3.1 and apply differential analysis as follows. For a test suite  $S$  with a system test  $T \in S$  and line coverage data  $C$  for  $T$ , we remove each line in  $C$  that is also covered by other tests in  $S$ . The result of this process is the subset  $U \subseteq C$  that is tested only by  $T$ . Alternatively, we calculate  $U$  so that  $U \subseteq C$  contains only these code fragments that are covered by the smallest subset of other tests. We map  $U$  to the corresponding functions within the source code, resulting in a set of functions  $F_u$ . We have to adjust for randomness, i.e., remove all functions that are known to exhibit nondeterministic execution. We identify such functions by comparing multiple coverage runs. The process results in a set of functions  $F$ . The cardinality of  $F$  depends on the system test characteristics. Based on our experience, the cardinality very low, typically reaching a value of 1. Fig. 6.2 shows an example where test 1 is the only test to cover fragment 6, in contrast to code fragment 2 that is executed by all tests. In this case, fragment 6 could be the core of test 1 and fragment 2 could implement functionality that is shared by all test executions, e.g., memory management.

Test	Source Code Fragment ( <i>SCF</i> )								
	1	2	3	4	5	6	7	8	9
Test 1	x	x		x	x	x	x	x	
Test 2	x	x		x	x				
Test 3	x	x	x				x	x	x
Test 4		x					x	x	x
Test 5	x	x	x	x	x				x

Figure 6.2: Example of code fragments executed by multiple tests. A green color with  $x$  indicates that a test executes the corresponding fragment. By differential analysis, we can identify that only test 1 covers *SCF* 6 and *SCF* 2 is covered by all tests.

### 6.1.4.3 Dynamic Test Data Extraction

We extract input and output data for a function call. More precisely, we monitor the values of arguments of a function as the execution calls the function and we monitor the return values. In either case, we will obtain memory content in a binary format. However, we aim to generate a developer-friendly presentation in terms of typical source code, i.e., instead of using a binary representation for memory state, we want to provide the source code that is required to obtain the desired state.

We propose the use of an adaptive approach for dynamic unit test extraction depending on the function arguments:

- Extract argument and return values using a debugger (Section 6.1.4.3) if the types are fundamental or enum.
- Extract argument and return values, and object members with a debugger if the types are object-type with only public members of type fundamental or enum and a default constructor.
- For other object types, apply time-travel debugging (Section 6.1.4.3) to trace object creation/modification.
- For member functions, use any of the previous methods to extract the corresponding object and dependent objects.

**Debugger:** A debugger allows investigating data and execution flow of a function without recompilation. We use *GDB* [237] which provides a convenient Python interface for automation and handles a wide range of C++ and *x64* specific behavior. We also developed tools for GDB which support additional features such as handling recursive function calls beyond the capabilities of the GDB *finish* command.

We use GDB without time-travel debugging for types fundamental or enum. For complex types, such an approach is not suitable. They may have private fields or constructors with additional dependencies. Tracking the creation of such complex objects with GDB leads to impractical overheads for our application. The overheads are due to the need for multiple test executions or extensive debug monitoring.

**Time-Travel Debugger:** A time-travel debugger enhances a debugger with the ability to execute a program backward. In practice, the execution does not strictly flow backward. Instead, state snapshots are used to simulate a backward execution. For our implementation, we use *UndoDB* [250]. UndoDB analyzes the binary that is executed and determines all sources of nondeterministic data such as system calls. These sources of nondeterminism will be captured during runtime. In addition, UndoDB creates copy-on-write snapshots of the program state during execution. UndoDB then allows reverting the execution flow to a historical state  $S$  of the process image. The program continues from  $S$  and executes all deterministic operations. In the case of nondeterministic operations, the captured events are replayed. When reverting to a program state between two memory snapshots, UndoDB identifies the closest snapshot and forwards the execution to the desired state.

UndoDB

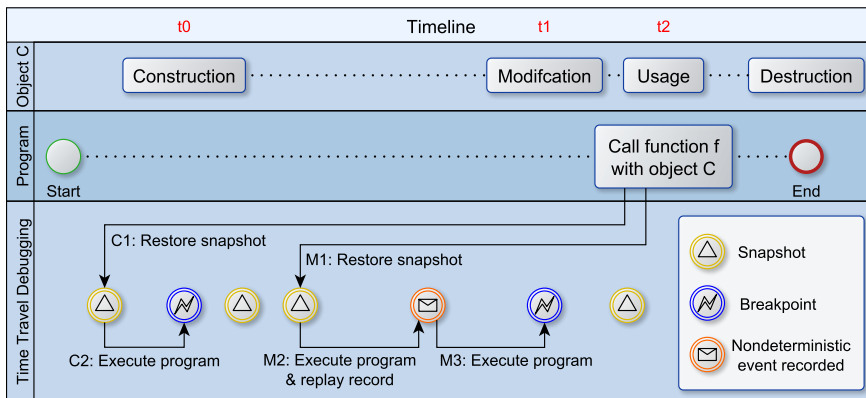


Figure 6.3: Using time-travel debugging to identify how object  $C$  is created and modified. In this example,  $C$  is created in steps  $C1/C2$ , and modified in steps  $M1$  to  $M3$ .

The authors of UndoDB state that nondeterministic operations represent only a small fraction of a typical software program execution [250]. Hence, UndoDB works even for large programs such as SAP HANA. UndoDB extends GDB [237], resulting in effortless migration between both tools.

Time-travel debugging provides an accurate and fast approach to identify the construction of complex objects because it allows following the execution flow backward until a point of interest. Fig. 6.3 visualizes the concept. Assume that the first argument of a function  $f$  is a complex object  $C$ . We start reverse execution at time  $t2$  corresponding to an invocation of  $f$ . We watch the memory address  $M1$  of  $C$  for a write access backward in time.

Our watchpoint is triggered at  $t_0$ ; here the innermost frame is a constructor (static analysis (Section 6.1.4.1) provides this information). By comparing the state of  $C$  at  $t_0$  and  $t_2$  we detect that a member variable  $A$  has a different content. We go back to  $t_2$  and watch the memory address  $M_2$  of  $A$  for a write access backward in time until  $t_1$ . For a unit test, we combine the source code for the object construction at  $t_1$  with the source code for the object modification at  $t_2$  to create  $C_2$ , and call  $f$  with  $C_2$ .  $C_2$  is in the same state as  $C$  for the system test. The test oracle is either the result of  $f$  or the state of  $C$  after the function call.

**Alternative Approaches:** We also investigated other techniques to monitor a function during execution such as binary instrumentation frameworks, source code level instrumentation or switchable monitoring statements. Due to limitations in terms of runtime and compile time overhead we refrained from further considering them.

An *instrumentation framework* (e.g., Intel Pin [71, 180]) can modify the binary representation of a program and inserted statements to collect function arguments and return values. In general, such tools support modifications for all functions identified by a search pattern and the performance overhead can be small if used sparsely. However, for complex arguments, the setup process requires recursive dependency tracking which can lead to a large amount of instrumentation and considerable resource consumption overhead. In addition, a precise understanding of *x64* calling conventions and their register placement is required to monitor function calls.

Instrumentation framework

Another approach is *instrumentation on the source code level* before or as part of the compilation. This allows the compiler to optimize the modifications resulting in reduced performance overhead. In addition, it is not necessary to have a good understanding of calling conventions for functions, because the compiler transparently applies them. However, complex arguments can require several recompilations to recursively track their creation, which results in a large time overhead. For instance, recompilation times can range from seconds to a single full recompilation of SAP HANA, which requires on a system with 40 cores (3 GHz CPU clock rate) up to 3 hours.

Instrumentation on the source code level

A project can be enhanced with *switchable monitoring statements*, depending, e.g., on environmental variables. This avoids the re-compilation overhead and is likely to have low runtime overhead due to CPU branch prediction. However, we found considerable runtime overheads in performance-critical code sections due to different decisions for alignment and inlining. Multiple test reruns can still be required to identify the placement of all switches for gathering all required setup steps for complex objects.

Switchable monitoring statements

#### 6.1.4.4 Unit Test Creation

We convert the extracted data into unit tests via configurable templates. The only limitation we faced is the integration within the build system. Selecting the correct minimum amount of header files, dependencies, linking options, or test file placement, is currently done manually. Our work did not focus on this aspect and we argue that it could be solved by including all header files based on static analysis, linking the whole project to satisfy all possible dependencies, and using a single directory for test files.



We also filter all duplicates when we convert the extracted data into unit tests, i.e., we select only one test for each distinct combination of argument and return values. In an extreme case, this reduces the number of unit tests for a function of SAP HANA from 3 224 680 to 256.

#### 6.1.4.5 Test Suite Reduction

In this optional step, we apply coverage-based reduction, i.e., we keep only the subset  $ST$  of tests  $T$  such that  $coverage(ST) = coverage(T)$  and the cardinality  $|ST|$  is lower or equal to the cardinality of any other subset with the same coverage. More precisely, we utilize a heuristic solver for the minimum set cover problem [53]. For the example mentioned in Section 6.1.4.4 with 256 distinct tests, only 7 tests remain after this step.

#### 6.1.5 Evaluation

We investigate the following research questions (RQ):

- RQ13 What reduction of test costs associated with frequent system test executions can we expect by applying our technique?
- RQ14 How large is the overhead of recording data for unit test extraction during system test execution?
- RQ15 How does our approach compare to tests generated by symbolic execution (in particular, KLOVER [258]) in terms of line coverage?
- RQ16 To what degree the developers of SAP HANA accept the extracted unit tests, and what is their feedback?

##### 6.1.5.1 Environment

We use a workstation with 1 TiB RAM, 160 cores (2.10 GHz clock frequency), and Linux (SLES 12). The state of our study projects is as of 2018-08.

##### 6.1.5.2 RQ13 Test Cost Reduction

We analyze the potential test cost reductions in two ways:

1. We measure the test time reductions achieved by replacing system tests via suites of unit tests.
2. We measure the overlap between lines covered by different system tests.

**Test Time Reductions:** We analyze the potential test cost reductions on SQLite and SAP HANA. For SQLite, we measure the original time  $T_O$  for system test executions, and we measure the time  $T_U$  for the execution of the corresponding unit tests. We generate the corresponding unit tests for SQLite by our approach. We verify manually whether our unit tests reflect the content of the system test and remove all cases where not.

For SAP HANA, we estimate the potential test cost reduction by a comparison of existing system tests against existing unit tests. This is possible in 107 cases. Note that the existing unit tests are only proxies for the potential time savings, as we cannot assume that they cover the same functionality as the system tests. However, due to the large number of tests, we expect that the magnitude of our measurements is reasonable.

We also analyze a second scenario where we include the setup times for the system tests. They consist of compilation times and system preparation (e.g., installation and startup). In this *with setup* scenario, we measure the time  $T_C$  from compilation start until the end of a system test execution. The setup time for SQLite are very low, therefore we omit the results.

For each time measurement, we calculate the arithmetic mean of 20 single measurements. We report the ratios  $T_O/T_U$  and  $T_C/T_U$  in Fig. 6.4.

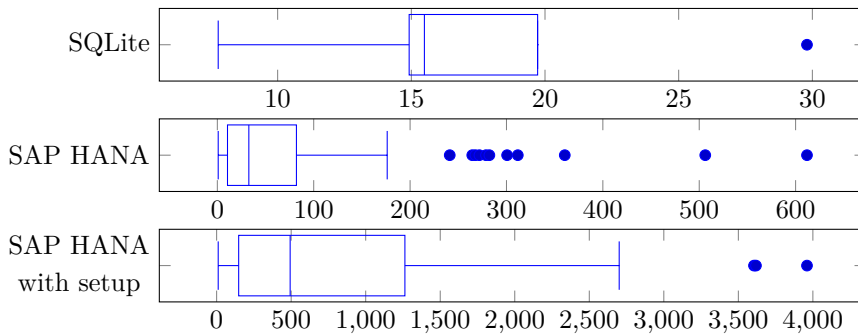


Figure 6.4: Potential test time reductions due to replacing system tests by unit tests (as ratios of system test execution time to unit tests execution time). Factors for SQLite based on extracted unit tests, results for SAP HANA based on existing unit tests.

For SQLite, the ratios are rather small. We attribute this fact to:

1. The small code size of SQLite with less than 150 000 lines of code. SQLite starts an empty database in less than a second.
2. SQLite TCL tests mix SQL system tests and unit tests.

It is therefore unclear, whether a project such as SQLite would benefit from our approach. We expect that for projects with a fast test suite, the test cost savings could be too small to justify the required time investment to apply our technique. However, such projects still benefit from other advantages of unit tests such as more precise error localization

For SAP HANA, the results reflect our initial assumption that system tests have considerably higher test costs compared to unit tests. The additional setup costs for system tests increase the total time cost by approximately a factor of 10. However, in practice, the factor depends on the configuration of the test framework and the testing strategy, which are both rather complex for large projects.

**Coverage Overlap:** To show that there is indeed a considerable overlap between sets of lines covered by different system tests, we use distance functions to calculate test execution similarities between two coverage data. As defined in Section 3.3.3, we use  $d_u$  (the number of lines executed by only one test) and  $f_s$  (the number of lines executed by both tests).

For SQLite, we consider the set of coverage data  $SC_S$  for all test suites represented by *.test* files, but we do not include fuzzy tests or tests only relevant for special configurations (testing nothing). For SAP HANA, we consider the set of coverage data  $SC_P$  for all system test suites. We calculate  $d_u, f_s$  for all elements in  $\{(x_i, x_j) \mid x_i, x_j \in M \text{ and } i > j\}$  where  $M$  represents  $SC_S$  or  $SC_P$ , i.e., we consider the “upper triangle” of the Cartesian product  $M \times M$ . We then report the minimum, maximum and arithmetic mean (AM). Table 6.2 presents the results.

Name	#	Covered Lines			$d_u$			$f_s$		
		Min	AM	Max	Min	AM	Max	Min	AM	Max
$SC_S$	721	8 670	11 626	19 822	40	3 974	11 833	8 058	9 639	17 677
$SC_P$	1 874	15 221	703 419	1 095 506	428	274 412	1 131 136	471	566 214	1 038 969

Table 6.2: Distance metrics for coverage data of all test suites  $SC_S$  (SQLite) and  $SC_P$  (SAP HANA).  $d_u$  represents the “unshared” part, and  $f_s$  the “shared” part.

For SQLite, all tests execute the same set of 8 085 lines, because, to our understanding of the source code, the test environment always starts an empty database. For both projects, the percentage of the same lines that are (on average) executed by all tests is fairly high (83 %, 80 %). This indicates a large number of redundant test executions and therefore potential test cost reductions. The minimal difference of two test suites is in both cases comparatively low indicating that it would also be possible to replace two system test suites by one system test accompanied by unit tests that cover the differences between the original two system tests. We conclude that both projects show the characteristic that system tests cover to a large fraction similar code and the test core may only be a small fraction.

— Answer [RQ13](#) —

For projects with a fast test suite such as SQLite, the possible test cost savings are comparably low with a factor less than 20. For SAP HANA, the possible test cost savings can be up to a factor of 1 000. For both projects, we expect that, in average, the test core is rather small and the redundancy in terms of executed lines is rather large. Therefore we expect considerable test cost reductions for both projects with our technique.

### 6.1.5.3 [RQ14](#) Overhead

We investigate 50 system test suites for SAP HANA and 879 test suites for SQLite. For each test suite, we measure the arithmetic mean of the execution times for 20 executions in three configurations:

1.  $T_O$  for the original configuration without modification,
2.  $T_S$  for our approach using regular debugger (see Section [6.1.4.3](#)),
3.  $T_T$  for our approach with time-travel debugging (see Section [6.1.4.3](#)).

We calculate the overhead factors  $F_S = T_S/T_O$  and  $F_T = T_T/T_O$  and present them in Fig. [6.5](#). For SQLite, the mean overhead factors are 17.86 for  $F_S$  and 136.56 for  $F_T$ . For SAP HANA, the mean overhead factors are 1.03 for  $F_S$  and 13.43 for  $F_T$ . Note that for SQLite,  $T_S$  is within a range of 0.30 s to 140 s and  $T_T$  within 0.60 s to 194 s.

Due to the lack of execution progress, we had to abort the measurements for  $T_T$  for SQLite in 1 case and for SAP HANA in 7 cases. This shows that time-travel debugging currently does not support all scenarios. The root causes remain unclear, we assume that frequent occurrence of nondeterminism resulted in large runtime overheads.

We did not measure the memory overhead. The methodology to measure varying memory usage within a shared memory system is very complex. Furthermore, we never experienced memory issues during our experiments.

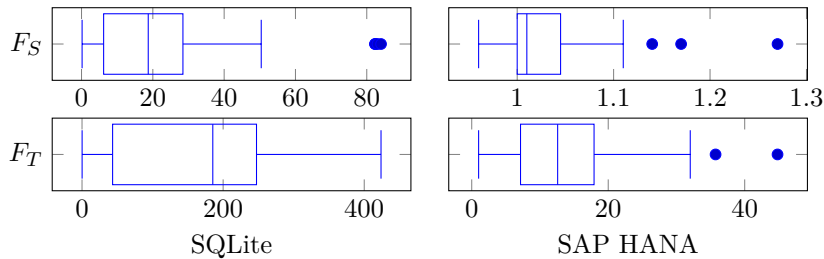


Figure 6.5: Runtime overhead factors for system test executions with recording via regular debugger ( $F_S$ ) and time-travel debugger ( $F_T$ ).

We identify the startup time of our approach as a major cause of the large factors of SQLite. Most of the SQLite test suites run in less than a second, therefore a fixed time for the startup can considerably increase the overhead factors even when the execution time of the tests is similar.

For SAP HANA, the small overhead factors for  $F_S$  follow our expectations for GDB with hardware breakpoints as they should have a negligible overhead to the execution time of the application. For  $F_T$ , the overhead is still in a practical range for large projects. However, the large standard deviation and several timeouts indicate that the behavior of the test has a large influence on the actual overhead.

— Answer [RQ14](#)

Dynamic unit test extraction via time-travel debugging leads to an average factor for runtime overhead of 13.43 for SAP HANA. For SQLite, this average factor is 136.56. Nevertheless, the approach is still feasible as test extraction is performed rarely.

#### 6.1.5.4 [RQ15](#) Comparison against KLOVER

KLOVER is one of the few recent research tools for to automatically generates tests in C++ [258]. However, the purpose is different to our approach as we aim to extract unit tests and KLOVER generates new unit tests via symbolic execution. However, we still compare to KLOVER in terms of numbers of lines executed by the generated tests (i.e., line coverage) too obtain additional information on how both approaches handle complex situations

We identify several files of SQLite where the line coverage of KLOVER is reported ([258], Fig. 4, values extracted automatically [221]). We select the files `hash.c` with the highest and `vdbe.c` with the lowest manual coverage (we skipped `random.c` because unit tests for randomness require a different methodology). In addition, we also analyze `complete.c`.

We manually setup memory management. SQLite dynamically initializes memory management functions such as `malloc` during startup and fails with nullpointer function access if they are not initialized. We categorize this as a static access, a limitation described in Section 6.1.6. We expect that the authors of KLOVER implemented the same adaptation.

For each file, we identify the set of functions by static analysis. For each function  $f$  in a file, we find all tests  $T_f$  that test this function by using our test core approach. Then, we execute each test in  $T_f$  and dynamically extract unit tests for  $f$ . We skip the test reduction step. We only investigate the final line coverage which is not changed by the coverage based test reduction. Fig. 6.6 reports the results for all study cases.

File	Line Coverage [%]		
	KLOVER		
	Manual	Automatic	Our Approach
complete.c	82%	56%	78%
hash.c	98%	65%	100%
vdbe.c	1%	2%	1%

Figure 6.6: Results reported by the authors of KLOVER [258] and our approach.

In the case of `complete.c`, the unit tests extracted by our approach covered all executable lines except for a function with a void pointer argument: function `int sqlite3_complete16(const void *zSql)`. We do not have information about the void pointer and reverse execution lead to the external TCL library where our analysis aborted. In fact, the argument is extracted by a TCL function call: `Tcl_GetByteArrayFromObj(objv[1], 0)`. Our implementation does not support such external code.

In the case of `hash.c`, our approach creates a large amount of tests for `unsigned int strHash(const char *z)`, a function calculating hash values. This set of tests likely has limited value because the distribution of return values is a more interesting property to check for a hash function than verifying hashes of individual inputs.

For `vdbe.c`, our approach was not able to extract the state for the arguments of the `sqlite3VdbeExec(Vdbe *p)` function that represents a large part of this file. The argument is a rather complex object that represents a virtual machine with its own state and contains the state of the global database via a pointer. We extracted several instances of this object that passed a manual inspection for correctness but resulted in segmentation faults when executed. We assume that KLOVER was also not able to generate this complex object, which would explain the low coverage of KLOVER.

We are not able to identify the strengths and weaknesses of our approach against KLOVER in terms of line coverage due to the unavailability of the source code for the tests generated by KLOVER. However, we expect that in all cases when we successfully extract the state of arguments and therefore call a function, we benefit from a large system test suite and can reach a high coverage. In contrast, KLOVER not only has to construct an argument to be able to call a function but must also resolve all conditions and branches within the function to reach a high coverage.

*Answer RQ15*

For source code covered by system tests, our approach results in higher coverage compared to KLOVER when arguments can be successfully monitored. However, there are also cases where we are unable to extract unit tests with our approach resulting in lower coverage compared to KLOVER.

#### 6.1.5.5 RQ16 Developer Acceptance and Feedback

Developers of SAP HANA used a prototype implementing our approach to extract 789 distinct unit tests for multiple C++ files. These unit tests were submitted (as source code) to the central repository where a mandatory code review is required for acceptance. All 789 unit tests were accepted by developers with a positive code review.

Since we do not focus on a detailed user study, we only briefly summarize the feedback from code reviews. Some developers appreciated the number of tests, some favored to reduce the number of tests with a test suite reduction approach (see Section 6.1.4.5). Some developers pointed out that the extracted tests should be extended with more input values based on their code understanding. We conclude that the extracted unit tests can provide a better understanding of the test coverage compared to system tests. Some developers have shown interest in applying our tool to more functions.

Based on the acceptance rate and feedback, we assume that developers generally accept maintainable unit tests with test oracles that provide additional unit test coverage if they do not require additional effort.

*Answer RQ16*  
 Developers of SAP HANA accepted all of 789 extracted unit tests. The corresponding code reviews provided positive feedback.

### 6.1.6 Limitations and Threats to Validity

#### 6.1.6.1 Conclusion validity

Due to the size of the projects, we decided to evaluate only samples. A more comprehensive study is required to improve the confidence in our results. However, the existence of results shows that our approach is practically feasible, it is only unclear to which extent. In Section 6.1.8, we discuss the work of Křikava and Vitek where they show the feasibility of dynamic unit test extraction in a large evaluation for R.

#### 6.1.6.2 Construct validity

Our definition of a test core may lack generalization. Based on discussions with our industry partner, there are cases where the results for test core identification are confirmed unanimously, but in other cases, there are discussions whether error checking code, utility code or pre/postprocessing code should be involved. Therefore, the final decision of whether to displace a system test is currently done by developers.

#### 6.1.6.3 Internal validity

Due to test flakiness, we are unable to evaluate the effectiveness of our approach in terms of failure detection rate. When we analyze the failure detection rate on historical data, the analysis is biased by a large percentage of flaky system tests. For SQLite, we found a flakiness rate of 0.60% for test executions as we measured execution times. The flakiness rate for SAP HANA is also larger than 0. Therefore, we would not know whether a test failure indicates flakiness or limitations in our approach. However, we argue that our approach does not reduce the quality for the final product due to the multiple testing stages.

To the best of our knowledge, mutations would not provide further insights, because when a system test execution can detect a mutation in a function  $f$ , our extracted unit tests for  $f$  can detect the mutation, too.

Our current implementation does not support all the functionality provided by C++. The most important issues are:

1. We do not fully support calls to static methods that change global state, such as file system read and write calls or network read and writes. For manual unit tests, the typical solutions in C++ involve rewriting the source code or introducing a completely new implementation with preprocessor or a linking order approach. It is unclear to us how to provide a general automatic and maintainable solution for this problem. Related work also report this problem as unsolved.
2. We do not fully support implicit C++ code such as generated constructors or several variants of template instantiations.
3. Pointers with void types require manual intervention. When a function requires a pointer to a memory region as an argument, the caller typically provides a second argument *size*. This is a frequent pattern in C code but also appears in C++ code. We may identify simple cases with a heuristic. However, complex cases require human intervention.
4. We do not support several specific corner cases or surprising usage of the C++ standard [138], i.e., code that uses infrequently used idioms. We attribute their presence to the size and age of the studied industry project.

System tests have several purposes. It is unclear whether our approach would be beneficial for, e.g., performance, security, or usability tests.

We cannot prove the correctness of the extracted tests. We assume that a successful system test execution implies a correct behavior of the called functions. However, two functions could produce wrong results, but the combination of them might mask the failure in a system test. Fault masking is a widely known phenomenon, but we did not encounter such a case.

#### 6.1.6.4 External validity

SAP HANA has several specific properties. It is

1. a database system,
2. an industrial project,
3. a large project,
4. tested by a large set of system tests.

The database context could affect the expected cost savings because software in other areas might have a different proportion of system tests to unit tests (see Section 6.1.7). The incentive for developers to minimize time and effort for test writing (and so prefer to code system tests instead of unit tests) could be affected by the industry setting. However, we observed a similar pattern in SQLite. For small projects, where all tests execute in less than a minute, further test cost reduction might not be beneficial. Finally, our approach requires an existing suite of system tests.

#### 6.1.7 Generalization and Utility

We expect that our approach or parts thereof can be used in other programming languages. Related work already targeted Java, R or FORTRAN.

Our approach can considerably reduce test costs when system tests execute to a large degree some common code (e.g., startup/teardown code). Database systems show this characteristic, but it is also present in compilers, GUI applications, or data analysis software.

Dynamic unit test extraction can also be useful in cases where no system tests exist, but any other form of execution is possible. For example, a developer can manually execute the program and extract unit tests for a specific part with less time effort than manual unit test creation.

We expect that dynamically extracted unit tests can be used as input for other techniques such as test case amplification, test case reduction, program analysis, or automated fault localization. Differential analysis on coverage data or time-travel debugging could improve other research techniques in terms of accuracy and execution time.

### 6.1.8 Related Work

Previous work on extracting behavior of system tests use terms such as test extraction, carving, capture and replay, or codelet extraction.

Křikava and Vitek propose automatic unit test extraction for R [167], which is to our knowledge the closest related work. Their tool *Genthat* extracts unit tests from execution traces of R programs provided by existing tests and so-called *vignettes*, runnable examples within the documentation of R packages. They instrument functions to record arguments and return values which are used to create source code for unit tests. *Genthat* extracts  $\approx 1.30$  million unit tests (26 838 without redundancy) for 1 545 R packages. We improve their work by reducing the problem size with a coverage based analysis and the utilization of time-travel debugging to recreate the source code for objects and their state. The authors also state that it is unclear “whether (their) approach would yield reasonable results in C or C++ (...)”

Elbaum et al. use the term test carving to extract *differential unit tests* [80, 81]. Their approach captures the program state immediately before ( $s_b$ ) and after ( $s_p$ ) the execution of a target unit  $m$ . They instantiate  $s_b$  and compare the modified  $m$  with  $s_p$ . Their approach serializes objects into XML [33]. Such a representation has low developer acceptance in industrial projects due to maintainability issues. Our approach recreates source code required to reproduce the state of objects, and we support C++ instead of Java. Also, we apply differential analysis on coverage data to reduce the problem size.

Saff et al. propose automatic dynamic test factoring [224, 225]. Given a system test that executes a subsystem  $T$  and interacts with other components  $E$ , their approach generates unit tests for  $T$  and introduces mock objects for all interaction with  $E$ . For this, they record a transcript that contains all procedure names and their arguments and return values. After the subsystem  $T$  is changed to  $T'$ , the transcript is used to verify that the behavior did not change for any interaction of  $T'$  with  $E$ .

Orso and Kennedy propose to record and replay all interactions with a subsystem of interest within a software program [207]. They employ an elaborated algorithm to reduce the size of the captured state by recording only these parts that are required for the replay and recoding events that recreate state to further reduce the state size. Joshi and Orso report on the *SCARPE* tool that implements this technique [146]. In comparison to their work, we target methods proposed by coverage based differential analysis and we aim to recreate source code for the unit tests instead of serialized object states and events. In addition, we support C++ .



Hovy and Kunkel propose a technique to generate unit tests for legacy FORTRAN code [123]. Their technique extracts data while running the original application and uses this data for test inputs. This requires the modification of the original application based on a static analysis with regular expressions. Our approach uses precise information provided by our compiler plugin and does not modify the source, therefore avoiding recompilation. In addition, they state as a limitation that “we don’t believe that our analysis approach with regular expressions would work for C or C++”. For objects, they use serialization (although they consider “plain FORTRAN”) while we recreate the C++ source code for the object creation.

Castro et al. propose *CERE* [42], a sophisticated tool for performance benchmarks and optimizations that supports all languages supported by LLVM [171]. *CERE* extracts a codelet, a specific part of the source code such as a function, and creates a dump of the memory and cache during program execution to allow codelet replay. Altogether, their approach and implementation is promising but creates a binary state and LLVM-IR. Although the textual IR could be an option for readable tests, it would introduce a LLVM dependency and create additional complexity for translations between LLVM-IR and C++ . Lee and Hall proposed a technique similar to *CERE* but less sophisticated [172]. Kim et al. report on *KGEN*, a Python tool that extracts FORTRAN kernels to standalone executables [157].

Jaygarl et al. propose to capture objects during execution and mutate the states of these objects for test generation with high coverage [142]. Their tool, *OCAT*, serializes objects in a global map during test executions and mutates these objects in later steps to reach a high coverage level. Our approach targets the extraction of maintainable unit tests and does not aim to generate new tests. *OCAT* could be used to extend our approach.

*KLOVER*, developed by Fujitsu, targets C++ [177, 258] and extends the basic concepts of *KLEE* [36] (a tool for C). They use symbolic execution to automatically generate tests with high code coverage. Symbolic execution solves constraints that are required to reach specific parts of the source code. However, such tests lack a test oracle. Therefore, such tests typically search for the violation of implicit test oracles such as assertions or exceptions. Our approach aims to generate unit tests with test oracles derived from system tests. However, both approaches could be combined, i.e., the results of our approach could provide inputs to improve the results of symbolic execution.

Yoo and Harman provide a survey of techniques to reduce regression test costs [257] and Kazmi et al. for regression test case selection techniques [149]. Garg et al. investigate concolic execution for C/C++ [98].

### 6.1.9 Conclusions

We propose dynamic unit test extraction to reduce the cost of regression tests in large C++ projects. Our approach combines several techniques, namely differential analysis of coverage data to reduce the problem size, time-travel debugging for accurate and fast extraction of complex object states, and multiple-stage testing strategy for safe test cost reduction. We address several limitations of previous research, such as limited maintainability of unit tests, limited support for large projects or missing C++ support.

Our evaluation shows that there is a large potential for test cost reduction with our technique. The runtime overhead of time-travel debugging is feasible, and the code coverage of extracted tests can be higher compared to other state-of-the-art tools. After applying our techniques to SAP HANA, developers accepted in code reviews all of 789 extracted unit tests.

There are several directions for future work. Additional engineering work is required to support more practical use cases, as a large project and C++ in general provide complex challenges. A comprehensible empirical study is required to better understand the strengths and weaknesses of our approach compared to other existing work for C++ or for Java.

## 6.2 Object Creation

Unit tests in object-oriented programming languages must instantiate objects as an essential part of their set-up. Finding feasible method-call sequences for object creation and selecting a most desirable sequence can be a time-consuming challenge for developers in large C++ projects. This is caused by the intricacies of the C++ language, complexity of recursive object creation, and a large number of alternatives.

To address this problem, we propose an approach that supports developers by suggesting code with desired characteristics. We confirm the significance of the problem by analysis of 7 large C++ projects and a survey with 143 practitioners. Based on gathered data we design an approach for recommending optimized method-call sequences for object creation. Our approach exploits accurate and efficient compiler-based source code analysis to build an object dependency graph used for graph traversals.

Survey with 143 practitioners

An evaluation on a large industrial project shows that, given criteria collected from developers, our tool proposes method-call sequences with a higher or equal ranking for 99% of 1104 cases compared to manually crafted solutions. Developer feedback and manual analysis confirm these results. Moreover, sequences proposed by our approach are considerably shorter than those found by approaches from previous work.

### 6.2.1 Introduction

In object-oriented programming languages [41, 52, 100], unit tests need to initialize objects under test (or objects used as parameters) as an initial part of the test set-up. This is typically achieved by appropriate sequences of method-calls that create and mutate such objects. In programming languages with a rather strict type system like C++ , it is already a challenge to find suitable method-call sequences to only *create* an object. For example, constructors in C++ may be private and only accessible to friend classes. Constructors might have parameters that require recursive creation of additional objects. Such dependencies can give rise to complex sequences with many intermediate objects. Moreover, there can be a significant variety of such sequences as for each (intermediate) object *multiple* constructors and/or (static) factory methods might be available.

Developers must analyze the *dependency hierarchy* for a targeted object to (i) find out a set of feasible method-call sequences (in short *sequences*) for

Dependency hierarchy

creating it, and then (ii) select an “optimal” sequence, if multiple options exist. Focusing only on instantiating objects, we call the challenges (i) and (ii) the *object creation problem* (OCP). For (i), it is not required to find *all* feasible sequences, as typically a subset is sufficient for designing a unit test.

Object creation problem

In the context of unit testing, a wide range of research work proposed techniques for generating method-call sequences to set-up objects into a desired state [92, 142, 182, 209, 228, 247, 249, 268]. In addition to these *sequence generation* methods, there are also *direct construction* methods [32]. These work focus primarily on achieving a desired object state, assuming that finding suitable code for object creation is trivial. However, through our discussions with developers in SAP HANA, we noticed a high development effort for solving the OCP. They have reported that finding and implementing object creation code as a part of unit test writing consumes a considerable amount of time. Interestingly, setting up an object state after its creation was typically considered a simpler task by these developers. We attribute the neglecting of the OCP in the literature to the relatively small project sizes that are used in evaluations and to the prevalence of other programming languages such as Java or C# in these previous work.

Related work assumes that the OCP is trivial

In this work, we first study the significance of the OCP and characteristics of suitable solutions by analyzing the code of large C++ projects and by conducting a survey with practitioners. Based on these results, we propose and evaluate an approach that suggests sequences of method-calls for object creation in C++ according to desired criteria. Our contributions are in detail:

- Confirming the significance for the OCP in 7 large C++ projects via static code analysis, and via a survey with 143 professional developers.
- Characterizing preferences of developers for object creation code based on the above-mentioned survey.
- An approach for suggesting method-call sequences for object creation that considers the identified preferences.
- An evaluation on 7 large C++ projects analyzing the effectiveness and demonstrating the improvements compared to previous work.

## 6.2.2 Motivation

The OCP can be difficult in C++ even for short programs. Large projects are likely to have recursive dependencies that need to be fulfilled when creating complex objects. With several constructors at each recursion level, the number of options for creating such objects can be huge. Consequently, the difficulty of OCP is likely to increase with project size.

### 6.2.2.1 Constructors

In Fig. 6.7 we can create an object of type `CA` by calling the public constructor. We can instantiate `CB` by calling the public default constructor, which is implicitly generated ([138], Clause 15). Hence, a developer must know the criteria for implicitly-defined C++ default constructors, which can be non-trivial in some cases. `CC` shows such a case. The object cannot be created with normal language constructs. `CC` has no default constructor (the member `a` is a reference), and list-initialization is not possible (`a` is private) [138].

```

1 class CA {
2     public:
3         int a;
4         CA(int a)
5             : a(a) {}
6 };
7
8 struct CB {
9     int a;
10 };
11
12 class CC {
13     int& a;
14 };

```

Figure 6.7: Three simple classes.

### 6.2.2.2 Derived Class

In Fig. 6.8, an object of type `Base` cannot be created directly due to the non-public visibility of the constructor and absence of a default constructor [138]. However, inheritance allows us to use `Derived` for `Base`. We have two options to create an object of type `Derived`. We might select the `Derived(int)` constructor because the second constructor depends on additional objects.

### 6.2.2.3 Factory Pattern

In Fig. 6.9, we are unable to create an object of type `P` directly as the only constructor is private. However, the class `P` declares `PF` as a friend. Therefore, the private constructor of `P` can be called from `PF`. Hence, to create an object of type `P`, we must discover the friend relationship to `PF`, identify and call `PF::createP()`, and recognize that the return type, a smart pointer, provides access to the desired object.

### 6.2.2.4 Summary

The wide range of patterns in C++ to provide objects can result in a large search space for developers that require an object of a specific type. Even more in large projects, a desired object can require additional objects as dependencies which further increases the search space. Therefore, automated recommendations for object creation can support developers.

## 6.2.3 Collecting Data from Users

We describe here the conducted interviews and a survey.

### 6.2.3.1 Methodology

**Exploratory Interviews:** We conducted the following experiment with 3 developers for 90 min each. In the first half, we observed the developers while they create real-world unit tests. We noted their steps, and we qualitatively assessed the amount of time required by each step. In the second half, we interviewed the developers whether our observations were correct. Based on this data we created a list of their distinct activities with associated (relative) time requirements. The survey described below uses this list. We omit further descriptions of the interviews because they were only explorative while the survey provides the empirical foundation for our work.

**Survey:** We designed an electronic survey [163] and conducted a trial run with 10 participants. Based on the results of this trial run and discussions with the participants we selected the specific formulations of the questions and the scale of the rating. For example, the trial group preferred the  $-3 \dots +3$  rating scale over an initially proposed ranking.

The electronic survey contains two questions shown in Table 6.3. Each question has multiple items with 7-item rating scales and a free text box for additional comments. Table 6.3 shows all items of the second question based on the experience of our industry partner and related literature [92, 234]. The first question has as items 10 steps of the test creation process that we derived from information gained by the interviews:

```

1 class Base{
2     protected:
3         Base(int m) { /*...*/ }
4 };
5
6 class Derived : Base{
7     public:
8         Derived(int n) {
9             /*...*/
10        }
11        Derived(ClassX& x) {
12            /*...*/
13        }
14 };

```

Figure 6.8: Object creation and inheritance.

```

1 class P {
2     friend class PF;
3     P(int id) { /*...*/ };
4 class PF {
5     public:
6         PF(int id) : id(id) {}
7         PF(ClassX& x) : id(x.id)
8             { /*...*/ }
9         unique_ptr<P> createP() {
10            return unique_ptr<P>(
11                new P(17));
12 };

```

Figure 6.9: Object creation via simplified factory pattern.

1. understanding of the source code,
2. necessary refactoring of the source code to make it testable,
3. conceiving test cases, i.e., thinking about possible input data and test oracles,
4. object creation/instantiation,
5. object state preparation,
6. mock creation,
7. writing test code (including test framework/build system code),
8. refactoring of the test code,
9. compilation/linking of test code, and
10. executing and testing test code.

### 6.2.3.2 Survey Participants

Our target audience is professional C++ developers. Our industry partner sent the survey to 1 185 recipients across multiple global C++ development units. We assume the most participants are from Germany, North America, and Asia, in this order. Due to concerns related to European privacy laws, we have no further knowledge about the cultural distribution and experience.

1185 survey recipients

### 6.2.3.3 Results

We received 143 responses, yielding a participation rate of 12%. SAP estimates that 50% of all 1 185 recipients regularly write C++ code, resulting in a relevant participation rate of 24%. Not all recipients rated all items, therefore the number of ratings varies between 116 to 133. 15 participants used the free text box for the first question, and 6 for the second.

Table 6.3 presents the results. For the first question we ordered the 10 items (the steps of a test creation) by the mean of the estimated time effort in the responses. This yielded the following ranking of the items in descending order: 2, 6, 1, 4, 3, 7, 5, 8, 9, 10. The step “object creation” is the fourth highest rated item. Table 6.3 shows distribution of responses. Even higher rated were steps (descending in this order): code refactoring 2, mock creation 6, and understanding of the source code 1.

For the second question, the highest ranked criterion is “minimal dependencies”, whereas “first working solution” is ranked lowest. Interestingly, even though mutability simplifies the state preparation, the criteria “objects should be mutable” is ranked relatively low.

The free text boxes for the second question contained mostly opinions about testing in general. Two valuable remarks are:

- Multiple product lines require additional effort, and
- Avoid objects that change the global state.

#### *Interview and Survey Results*

Professional developers in large C++ projects consider (a) time effort for implementing object creation as high, and (b) the minimal amount of dependent objects as the most important criterion for selecting a method-call sequence for object creation.



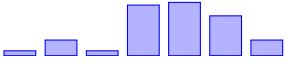
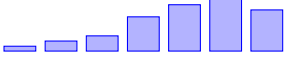

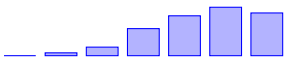
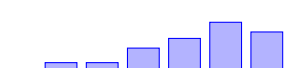
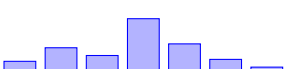
Items	Rating	Mean	Median	<i>n</i>
<b>Question 1:</b> Which aspects of implementing unit tests in SAP HANA require considerable time effort?				
4) Object creation/instantiation for source code under test.		0.83	1	133
	low time effort -3 -2 -1 0 +1 +2 +3 high time effort			
<b>Question 2:</b> If there are multiple options for object creation within a unit test (e.g., several constructors, factory methods), which criteria are important to select one option?				
1) The amount of dependent objects should be minimal, e.g., constructors with fewer additional object-type arguments are preferred.		1.73	2	120
2) The state of objects should be as mutable as possible to modify the objects during tests.		0.40	1	119
3) The object is created in the same way at other places in the productive source code.		1.09	1	120
4) The object is created in the same way at other places in the test code.		0.60	1	119
5) The objects should be related to the code under test, i.e., the distance between code under test and source code for the object should be minimal.		1.48	2	119
6) The objects should not be complex, i.e., metrics such as the cyclomatic complexity or coupling should be minimal.		1.15	2	119
7) It should work at all, i.e., the first working solution is good enough.		-0.13	0	116
	not important -3 -2 -1 0 +1 +2 +3 very important			

Table 6.3: Survey questions and results. Column *n* indicates how many participants have rated the corresponding item.

### 6.2.4 Approach

We describe here our approach to solve the object creation problem. Fig. 6.10 shows an overview of our method.

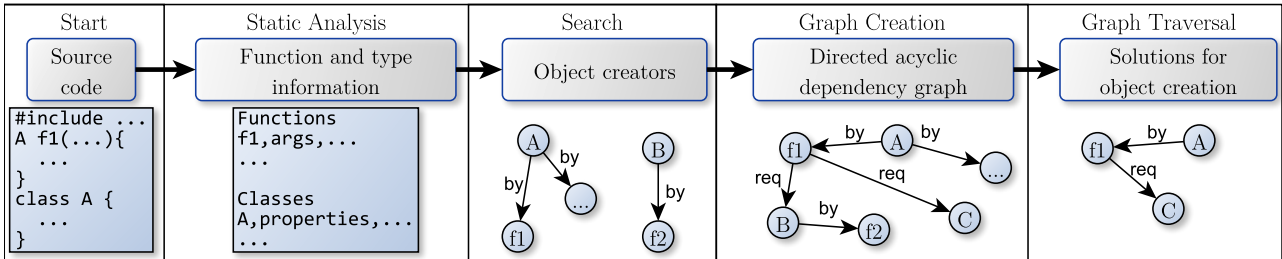


Figure 6.10: Approach overview.

The *object creation problem* (OCP) asks to find a method-call sequence which instantiates an object of a desired type  $T$  such that the sequence satisfies or optimizes given criteria (see Section 6.2.3). As mentioned, we do not consider mutating the created object into a specific state.

An object in C++ is typically created by a constructor ([138], 4.5). In simple cases, such “object creators” have no parameters, and the OCP is easy to solve. In other cases, we may have to instantiate multiple parameters of a creator and have to recursively solve the OCP for each of them.

This process can unfold in different ways, yielding alternative method-call sequences, each able to instantiate an object of a desired type. Such a sequence is termed a *valid (method-call) sequence* or just a *sequence*. Without explicitly enumerating all valid sequences, we search for one which optimizes certain criteria (see Section 6.2.3). We call such a sequence a *solution*.

Object creation problem

Sequence

#### 6.2.4.1 Object Creators and Dependency Graphs

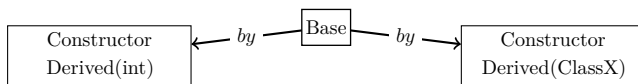


Figure 6.11: Top nodes of a dependency graph for Fig. 6.8.

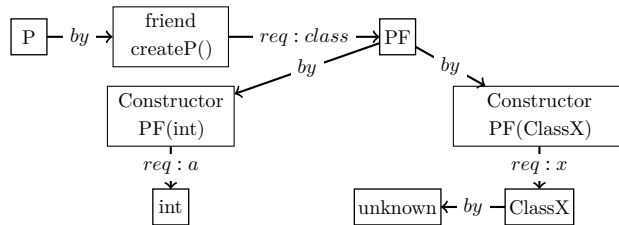
For a given type, let  $C$  be the set of all functions which provide an instance (object) of this type, e.g., constructors or methods (see Section 6.2.4.2). We call such a function an (object) *creator* for a given type. In Fig. 6.11, type PF has two creators: constructors PF(int) and PF(ClassX).

Creator

A *dependency graph*  $G$  for a desired type  $T$  is a directed acyclic graph with the following properties. All nodes of  $G$  correspond to either types or object creators. For a node  $v_t$  corresponding to a type  $t$  and each creator  $c$  of  $t$ ,  $G$  has a node  $v_c$  (corresponding to  $c$ ), and an edge  $v_t \rightarrow v_c$  labelled with “by”. For each node  $v_c$  and each required parameter  $p$  of  $c$ ,  $G$  has a node  $v_p$  corresponding to a type of the parameter  $p$ , and an edge  $v_c \rightarrow v_p$ . We label such edges as “required” (or “req” for short) and specify what is required, e.g., the parameter name or the class instance.  $G$  might also have special nodes *unknown* linked from each type without any creator. Finally, there is a (root) node corresponding to the targeted type  $T$ . Given  $G$ , we define the *size* of  $G$  as the number of ‘type’-nodes. Fig. 6.12 shows a dependency graph for a targeted type  $P$  from Fig. 6.9 with size 4.

Dependency graph

Our approach has a (one-time) *preprocessing phase* to determine all creators for each type in the source code (Section 6.2.4.2). Given as input a desired type to be instantiated, we construct a corresponding dependency graph for this type (Section 6.2.4.3) and determine an optimal solution via graph traversal described in Section 6.2.4.5.



Preprocessing phase

Figure 6.12: Dependency graph for Fig. 6.9.

### 6.2.4.2 Searching for Object Creators

We determine all creators for any type available in the source code by static analysis. To this purpose, we implemented a plugin for Clang, a C++ frontend for the LLVM compiler infrastructure [171]. The plugin extracts all data during compilation resulting in accurate data even in the case of multiple compilation stages where or complex preprocessor directives where the final source code for compilation is generated by other source code generation programs. The data contains all identified object creators and additional information for them such as order and types of parameters. In the following, we describe which language concepts are analyzed.

Clang compiler plugin

For a given type  $T$ , let  $C$  be the set of object creators for  $T$ . To simplify the presentation, we (a) use the term *class* also for *struct* and *union* ([138], Clause 12)), (b) shorten type  $T$  or class  $T$  to  $T$ , and (c) do not distinguish between references/pointers to an object and the object itself.

**Constructors:** We add to  $C$  all useful constructors for  $T$  and consider them technically as functions. A constructor is useful if it has *public* visibility, no attribute *deleted* [138], and is no copy or move constructor [138].

**Inheritance:** We include in  $C$  all accessible constructors for any subtype (multi-level, acyclic ([138], Cl. 13)) of  $T$ .

**Factory Method Pattern:** We include in  $C$  all accessible static methods that return an object of type  $T$ . We ignore non-static methods. They are rarely useful because the underlying object must be created. They are only useful in the case of friends that we handle separately.

**Friends:** C++ allows a class to declare other functions or classes as *friends*. Friends can access private and protected functions and members of the class declaring the friend [138]. We add to  $C$  all methods that provide an instance of  $T$  and are marked as *friends*, or are defined in *friends*-classes of  $T$ .

**Smart Pointers:** A smart pointer is a data structure that wraps a pointer in C++ and provides additional functionality such as automatic memory management. We “shorten” the indirection of smart pointers ([138], 23.11) which seem to appear frequently in modern C++ code. E.g., for Fig. 6.9, we use  $P$  instead of `unique_ptr<P>`. We detect such smart pointers by identifying the standard implementation cases and by providing a list of custom implementations of the corresponding project.



**Output Parameter:** Objects can be created by an *output parameter* pattern where a parameter provided by the caller is modified by the callee.

Fig. 6.13 shows two examples of this pattern, *init1* and *init2*. The first uses *call-by-reference*, the latter *call-by-sharing*. It is challenging to accurately detect such cases in general. Considering all functions with adequate parameters would be misleading and requires further analysis. The general decision problem if a function execution instantiates a specific class is undecidable due to the halting problem [62].

We use a heuristic that reports a function if any parameter is of type  $T$  and the body contains a constructor call for  $T$ . However, due to false positives, we do not include such results in  $C$  automatically but report them to developers for manual investigation.

**Public Members:** A second class  $U$  can contain a public member  $M$  of type  $T$ . We do not consider using  $M$  to be an intended way to create/access a desired object and therefore ignore it. Also, for creating  $T$ ,  $U$  must use any of the detectable variants described before.

**Static Casts:** It is possible to create an instance of  $T$  by a static cast of another object. C++ allows such casts ([138], 8.4) and also allows a memory copy without further type checks (e.g., [138], 6.9, item 2). However, such an approach could create incorrect states of objects in memory [138, 183] and is therefore not considered in  $C$ .

#### 6.2.4.3 Constructing a Dependency Graph

Given the creators of all types, we can create a dependency graph for a type  $T$ . We start with a root node corresponding to  $T$  (node of level 0), and retrieve all creators of  $T$ . Each of them gives rise to a new 'creator'-node (level 1), and an edge between root and such a 'creator'-node. Then for every parameter  $p$  of a 'creator'-node  $v_c$  from level 1 we retrieve the type  $t$  of the parameter  $p$ , and create a (level 2) 'type'-node  $v_t$  together with an edge  $v_c \rightarrow v_t$  (see graph definition in Section 6.2.4.1). This process then repeats recursively for each newly created 'type'-node (on the levels 2, 4, ...) until any of the following stopping criterion is met.

**Stopping Criteria:** We investigate each  $v_t$  in levels  $\{0, 2, 4, \dots\}$ . We stop if any of the following conditions holds:

1.  $v_t$  has no creators.
2.  $v_t$  is fundamental, enum or function.
3.  $v_t$  is included in a white list (a list that contains types for which we pre-define solutions, e.g., types that may have a large number of creators but are in fact simple to create).
4.  $v_t$  is included in a list of objects to mock (if a mock object should be used instead of the original object).
5.  $v_t$  is defined outside of the software project.
6.  $v_t$  is already processed (to prevent cycles).
7. The corresponding parameter is unnamed or has a default argument.
8. The number of creators is larger than a predefined threshold, e.g., 100 (such a large list may contain a suitable creator).
9. The recursion depth is larger than a predefined threshold, e.g., 5 (we do not expect any practical results at such depths).

```

1 void init1(T** object){
2   *object = new T(5);
3 }
4 void init2(O2& object2){
5   object2.setT(new T(7));
6 }

```

Figure 6.13: Output-parameter.

#### 6.2.4.4 Valid Method-Call Sequences

The dependency graph for type  $T$  allows finding all method-call sequences which instantiate  $T$ . Note that there can be many such sequences for  $T$ . For example, in Fig. 6.8, there are two sequences: one using the constructor `Derived(int)`, and another using the constructor `Derived(ClassX)`.

In general, a valid sequence corresponds to a subgraph  $H$  of a dependency graph with certain properties:

1.  $H$  contains the root node.
2. For every included 'type'-node  $v_t$ ,  $H$  has exactly one child (a creator).
3. For each included 'creator'-node  $v_c$ ,  $H$  contains all child nodes.
4. All leaves ('type'-nodes without descendants) correspond to types which can be created trivially.

For practical use, we do not need to explicitly enumerate all valid sequences in a dependency graph. Instead, we can compute a single sequence that optimizes desired criteria, using the dependency graph as input.

#### 6.2.4.5 Finding Optimal Solutions via Graph Traversal

We describe an algorithm that finds a *solution* for the OCP of a type  $T$ , i.e., a method-call sequence optimizing certain criteria. Following the results of the survey in Section 6.2.3, our objective is fixed as the minimal number of dependencies, or “size” of a sequence (formalized below).

A solution is a subgraph of a dependency graph. Therefore, the size follows the definition given in Section 6.2.4.1 and is the number of 'type'-nodes. Fig. 6.12 shows that the (unique) valid sequence for  $P$  has size 3, and contains objects with types  $\{P, PF, int\}$ .

A solution is thus a sequence with a minimum size over all valid sequences. Listing 6.1 shows pseudocode of a divide-and-conquer algorithm to find a solution.  $ALG_o$  recursively finds an optimal subgraph and returns a solution (corresponding to a desired sequence) if it exists. Note that the algorithm can be easily modified to return multiple ranked recommendations.

```

1 def ALG_o('creator'-Node vc) // e.g., root of a DG
2   Node result = copy of vc
3   for each descendant t of vc # parameters
4     Node solution = 'unknown'
5     for each descendant c of t # c is creator of t
6       Node newSolution = ALG_o(c)
7       solution = minSize(solution, newSolution)
8     append solution to result
9   return result

```

While  $ALG_o$  minimizes the size of the output sequence, the function `minSize` (line 7 in Listing 6.1) can be replaced to use another criterion. Hence, the proposed algorithm supports different optimization criteria. In the evaluation, we assume the size of the sequence as the optimization objective.  $ALG_o$  has the following properties:

- It is a divide-and-conquer algorithm [53] by design.  $ALG_o$  recursively breaks down the problem of finding a dependency graph into finding dependency graphs of the dependent types until we find simple solutions. Such simple solutions exist because objects are typically composite types that (at some point) consist of fundamental types.

Solution

A solution is a subgraph of a dependency graph

Listing 6.1:  $ALG_o$  to find a solution for a given type. Function `minSize` returns the input with smaller size.

- We may trivially use multiple threads for parallel execution.
- The worst-case algorithmic time complexity of the algorithm is in  $\mathcal{O}(n^n)$  where  $n$  is the number of types within the project.
- Re-using results for recursion is in general not possible because each recursion depends on a state (the list of already visited types).
- We can apply a branch and bound approach [53] where we avoid descending recursion if we already know a better solution.

The time complexity may require additional explanation. For a project with  $n$  different types, we assume a worst case, that is all types depend on all other types. We analyze the steps required by  $ALG_o$  for a type  $T_1$  in such a case. For this purpose, we calculate the number of required operations for each level of the dependency graph, starting with the root level  $L_1$ . An operation is here the lookup of creators for a type (which we assume is constant). Then we can observe:

- At  $L_1$ , we only have  $T_1$  and, therefore, only one operation. We must create  $n - 1$  other types (each other type).
- At  $L_2$ , we have to investigate  $n - 1$  types. Hence,  $n - 1$  operations. For each type on this level, we must create  $n - 2$  other types (all except  $T_1$  and a specific  $T_2$  for each branch).
- At  $L_3$ , we have to investigate  $n - 2$  types  $n - 1$  times. Hence,  $(n - 1) \times (n - 2)$  operations. We must create  $n - 3$  other types.
- At  $L_4$ , we have to investigate  $n - 3$  types  $(n - 1) \times (n - 2)$  times. Hence,  $(n - 1) \times (n - 2) \times (n - 2)$  operations. We must create  $n - 4$  other types.
- At  $L_n$ , we have to investigate  $n - (n - 1)$  types  $\prod_{i=1}^{n-2} (n - i)$  times. Hence,  $\prod_{i=1}^{n-1} (n - i)$  operations.

Next, we sum over the operations on each level to obtain the total amount of operations as a sum over the product of a sequence:

$$1 + \sum_{j=2}^n \prod_{i=1}^{j-1} (n - i). \quad (6.1)$$

A proof by induction follows by the above examples for each level. To illustrate the formula via an example, we obtain for  $n = 3$

$$1 + \sum_{j=2}^3 \prod_{i=1}^{j-1} (3 - i) \quad (6.2)$$

$$= 1 + \prod_{i=1}^{2-1} (3 - i) + \prod_{i=1}^{3-1} (3 - i) \quad (6.3)$$

$$= 1 + \prod_{i=1}^1 (3 - i) + \prod_{i=1}^2 (3 - i) \quad (6.4)$$

$$= 1 + 2 + 2 \times 1 = 5. \quad (6.5)$$

Which is expected for 3 types because we have to investigate the first type at  $L_1$ , then both other types at  $L_2$  and finally, on each subgraph at  $L_3$ , the respective third left type. Hence, 5 operations in total.

Conclusively, based on Eq. (6.1), the worst-case time complexity of  $ALG_o$  is in  $\mathcal{O}(n^n)$ . However, in practice,  $ALG_o$  is rather efficient because the amount of dependent objects is typically not large and therefore the recursion width and depth are rather small for practical instances.

### 6.2.5 Implementation Details

We discuss aspects of our implementation that are tightly related to the specifics of the C++ language. Our approach does not rely on them but the results improve if we consider these technical intricacies. For each aspect, we refer to the corresponding section of the C++ 2017 standard [138].

#### 6.2.5.1 Object Definition

In C++, “an object is created by a definition (6.1), by a new-expression (8.3.4), (or ...) (and) has a type” ([138], 4.5), “that is not a function type, not a reference type and not cv void” ([138], 6.9, item 8). “A class is a type” ([138], Clause 12) and “a constructor is used to initialize objects of its class type” ([138], 15.1, item 2). “An object of a class consists of a (possibly empty) sequence of members and base class objects” ([138], Clause 12). “A *class-specifier* is commonly referred to as a class definition” ([138], Clause 12, item 2) and contains a *class-key* that is one of {class, struct, union}.

Therefore, we say that an object is an instance of a class and a class is either indicated by a keyword class, struct, or union. These keywords result in different default visibility ([138], Clause 14, item 3 and 14.2, item 2). However, other than that, class and struct are identical. Therefore, we only use the term class for class, struct and union.

#### 6.2.5.2 Templates

C++ *templates* define “a family of classes, functions, or variables, or an alias for a family of types” ([138], Clause 17).

Fig. 6.14 shows a function template. Compiler typically implement templates by creating a distinct class or method for each implicit or explicit instantiation of a template. Therefore, the number of different objects and functions in a C++ project can drastically increase by extensive template utilization. Explicit template instantiation can require the analysis of each instantiation because each explicit instantiation can implement different functionality. An analysis of only the abstract type would not provide enough information in such cases. Templates in central classes can lead to large dependency graphs due to duplicates of all functions inside a class.

Each template instantiation of function `func` in Fig. 6.14 provides a possible way to create an object of type *A*. `func` could be defined as a member function within a central class. In large projects, central classes have over 1 000 template instantiation, resulting in a large number of nodes within the dependency graph. Our implementation either tries to detect such cases and flags them as a single node or ignores nodes with too many options. In the case of several hundred options, they likely contain a variant that can be constructed without additional requirements. Hence, it is justified to avoid the explicit exploration of all options.

```

1 template<class U>
2 A func(U parameter) {
3     /* setup a of type A */
4     return a;
5 }

```

Figure 6.14: C++ function template example.

### 6.2.5.3 Pointers, References and Arrays

C++ uses *pointers* ([138], 11.3.1), *references* ([138], 11.3.2) and *arrays* ([138], 11.3.4). We argue that if we can create an object of type  $T$ , the usage of pointers, references and arrays does not add additional requirements. Hence, our Clang plugin replaces them with the resolved type.

### 6.2.5.4 Smart Pointers

*Smart pointers* ([138], 23.11), i.e., a generic data type wrapped around a raw pointer in C++ to manage the object the raw pointer points to, are frequently used in modern C++ software projects. Our implementation directly unwraps them, therefore removing one additional layer in the analysis. We argue that if we can create an object of type  $T$ , the creation of a smart pointer that wraps these objects does not add additional requirements.

### 6.2.5.5 typedef and alias-declaration

In C++ , a *typedef* declaration ([138], 10.1.3) or the semantically equivalent alias-declaration with the keyword *using* ([138], 10.1.3, item 2), can be used to provide a synonym for another type. Typedefs are most often used to provide a simple alias for complex type names. Multi-layer typedefs, i.e., a typedef for a typedef, are common in large projects. Resolving typedefs, that is, recreating the original type name, is required for a complete type analysis. Our Clang plugin resolves any typedefs to the ultimate root type.

### 6.2.5.6 Lambdas

C++ *lambdas* ([138], 8.1.5) are unnamed function objects. They are typically used in a non-static way, i.e., they cannot be referenced globally. We encountered only a very low number of cases where lambdas could be used in a static way, i.e., could be accessed from any part of a program. In such cases, the use cases for lambdas did not include object creation, therefore we do not consider lambdas as an additional source for object creation, but we support them if they fall into one of the existing categories we search for. Lambdas also appear as parameter types. We assume in general functions as a parameter type can be trivially created, e.g., with an unnamed lambda.

### 6.2.5.7 auto Keyword

C++ 11 introduced the generic type specifier *auto* ([138], 10.1.7.4), i.e., *auto* can be used instead of a concrete type and the compiler deduces the type. For a complete analysis, each instance of *auto* must be resolved to the concrete type which is done by our Clang plugin.

### 6.2.5.8 Multiple Return Values

C++ 17 introduced the concept of *structured bindings* ([138], foreword and 11.5). The main purpose of structured bindings is simplified handling of multiple return values. We only rarely encountered return values of the type tuple (less than 0.10% in our industry project), therefore we did not investigate them further. But we expect that the usage increases with the

addition of structured bindings within the C++ standard. It is unclear to us how to detect structured binding for a class type according to [138] 11.5, item 4. This would require a rather complex analysis of the class members.

#### 6.2.5.9 Constant Expressions

We support constant expressions, that are, “(expressions) that can be evaluated during translation” ([138], 8.20, item 1), as far as they compiler “removes” them. Our Clang plugin utilizes the Clang compiler that evaluates all constant expressions during translation. In our experiments, we encountered constant expressions very infrequently.

#### 6.2.5.10 Stopping Criteria for Recursive Graph Construction

In addition to the general criteria defined for dependency graphs, the recursive graph construction will not further inspect a function parameter if: *a)* The parameter is not named ([138], 11.3.5, item 13) and therefore not used inside the function. *b)* The parameter has a default argument ([138], 11.3.5, item 13) and therefore an external creation is not required. *c)* The parameter has only a single argument with the same type as the object we want to create. This happens frequently because of copy constructors and move constructors ([138], 15.8.1).

### 6.2.6 Evaluation

We investigate multiple research questions (RQ). First, we analyze the results of the search phase and characterize the studied projects (RQ17). Then, we study the existence of solutions (RQ18). Finally, we verify our solutions (RQ19), and compare our approach against related work (RQ20).

#### 6.2.6.1 Evaluation Setup

We searched for large C++ software projects on GitHub, related literature and publicly available lists. We filtered the list of possible projects based on the following criteria. The project

1. uses C++ as the main programming language,
2. supports compilation with Clang,
3. has more than 10 000 different object types (it is “large”).

The resulting list contains 7 projects as shown by Table 6.4, source lines of code (SLOC) measured by cloc [61]. More projects might fulfill these criteria, as we could not adapt some projects to compile with our plugin, and we pre-selected projects based on their expected number of types.

We use a system with 4 processors, 160 cores with 2.10 GHz, and 1 TiB RAM. The Clang plugin increases the compilation time by a factor of 1.20 and generates 27 GiB of data for SAP HANA and 0.20 GiB to 8 GiB for the other projects. For practical reasons, SAP HANA requires parallel compilation. This would increase the intermediate data size to 4 TiB. Therefore, we implemented lock-free duplicate filtering to mitigate this issue. We consider these overheads acceptable for practical purposes. The execution time of our algorithm  $ALG_o$  (Listing 6.1) is below a second. This is considerably

Project	#Obj. Types	#Functions	SLOC
SAP HANA 2018-11-24	735 194	4 210 541	11 065 382
Boost 1.66 [65]	30 548	53 528	4 392 925
CERN ROOT 6.13/08 [66]	100 705	730 507	3 417 362
Firefox 55.0.3 [68]	134 554	940 346	7 343 242
LLVM Clang 6 [69]	88 913	591 112	242 032
MySQL 8.0.11 [70]	54 360	199 692	3 791 989
ScummVM 2.0.0 [72]	13 527	148 350	1 830 628

Table 6.4: List evaluation projects.

faster than a manual search, which can require more than 10 min per case according to our observations during the interviews with developers.

#### 6.2.6.2 RQ17 Search Phase and Project Characteristics

RQ17 What is the variety of types and the distributions of object creators found in the search phase?

For each project in Table 6.4, we count the number of object types, the size of classes and record whether they have a default constructor (*DC*). For each object type, we count the number of creators.

**Object Types, DC, and Class Sizes:** We count each class as an object type. For class templates, we count explicit and implicit class template instantiations [138]. Table 6.4 presents the results. Table 6.5 shows for each class (a) the size  $n = \text{lineEnd} - \text{lineStart} + 1$  grouped into small ( $n < 5$ ), medium ( $5 \leq n < 50$ ), and large ( $50 \leq n$ ) as reported by our Clang plugin, and (b) whether it has a DC. Technically, a DC exists, if the existence is reported by the compiler and it is not marked as deleted [138].

Project	Size Groups [LOC]			
	0..4	5..49	50..∞	All
SAP HANA	83.64	63.82	52.91	68.98
Boost	96.00	82.88	69.15	85.92
CERN ROOT	88.15	81.18	59.31	79.76
Firefox	93.49	79.02	81.32	83.24
LLVM Clang	87.26	79.62	56.55	77.55
MySQL	94.56	89.40	68.91	87.98
ScummVM	97.37	61.69	69.42	70.83

Table 6.5: Distribution of default constructors. E.g., 94.56 % of all classes in MySQL with 1-4 lines have a default constructor.

**Object Creators:** For each object type  $T$ , we collect the set  $C_T$  of all creators and count  $|C_T|$ . In our dependency graph,  $|C_T|$  is the number of nodes connected with a *by*-edge from  $T$ . For Fig. 6.12, we can deduce that *PF* has 2 creators. Fig. 6.15 presents the results.

**Discussion:** Fig. 6.15 shows that the search phase finds at least one creator for 93 % to 99 % of all object types. Among all projects, Clang has the highest percentage of empty results (7 %). We conclude that the search phase provides reasonable results. A manual investigation of examples with empty solutions shows mainly cases where we were also unable to find solutions manually or where an object was used within a class-internal usage scenario. Further work is required to characterize such cases.

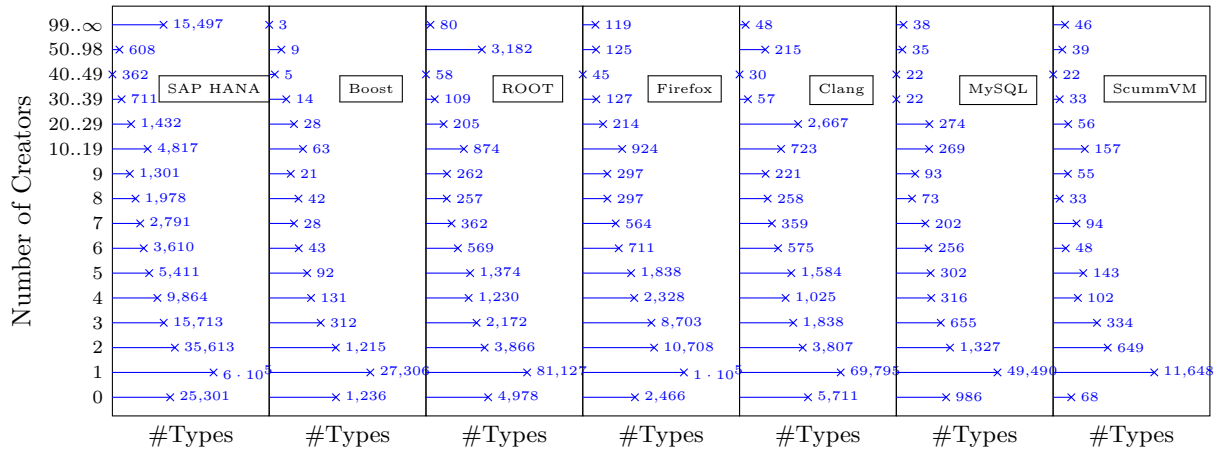


Figure 6.15: Histogram for number of object creators per type (x-axis in log-scale). E.g., for Boost, the search phase finds no creator in 1 236 cases and exactly 1 in 27 306 cases.

The most frequent cardinality is 1 due to the presence of DC in small classes. Table 6.5 shows the distribution of DC in different size groups of classes. It is rather common for small classes to have a DC. Large classes often do not have default constructors, but multiple creators. This indicates that our approach is more effective for large classes.

At least 6% to 20% of all object types and 16% to 38% of all large classes have more than 1 creator. This confirms the relevance of the OCP.

*Answer RQ17*  
 The search phase finds creators for 93% to 99% of all object types. The most frequent result is a single creator. For at least 6% to 20% of all object types, there exists more than one creator.

### 6.2.6.3 RQ18 Existence of Solutions

RQ18 What is the fraction of functions in studied C++ projects for which our approach can successfully find solutions for all arguments?

In practice, it is important to create all arguments of a function, which may require instantiating multiple different object types. Therefore, we switch our focus from objects to functions. In this section, we define a dependency graph (DG) of a function  $f$  as a union of dependency graphs for the types of each (required) parameter of  $f$ . Analogously, a valid sequence/-solution for  $f$  is the union of the respective concepts over all parameters of  $f$ . In other words, we introduce an artificial root node, consider  $f$  as an creator for this root and apply our approach accordingly.

**Functions:** Our Clang plugin reports all functions generated by the compiler. This includes static functions, object member methods, lambdas, and each function template instantiation. Table 6.4 presents the results.

**Size of Dependency Graphs (DG) and Solutions:** We apply our approach to each function and obtain a dependency graph  $DG$  and a solution  $S$ . Fig. 6.16 shows for all functions in each project the sizes of  $DG$  and  $S$ . Table 6.6 presents the percentage of functions  $\%Solved$  where our approach finds a solution and, over all functions, the average of the solution size  $|S|$ , the graph size  $|DG|$ , and the number of parameters  $|args|$ .



Project	%Solved	$\overline{ DG }$	$\overline{ S }$	$\overline{ args }$
SAP HANA	97.98	11.36	2.20	1.63
Boost	98.31	4.56	1.78	1.47
CERN ROOT	96.11	14.49	1.85	1.18
Firefox	97.16	15.16	1.95	1.39
LLVM Clang	94.74	18.96	2.03	1.28
MySQL	98.38	9.10	2.24	1.59
ScummVM	99.89	5.19	1.56	1.17

Table 6.6: Results for all functions. Solved percentage and the means for dependency graph sizes (DG), solution sizes (S), and amount of arguments (args).

**Discussion:** Table 6.6 indicates that our approach finds full solutions to create all arguments for 94% to 99% of all functions. Fig. 6.16 shows that the solution size (Section 6.2.4.5) is typically rather low compared to the size of  $DG$ . The average solution size  $\overline{|S|}$  is slightly larger than  $\overline{|args|}$ , which is expected. Unnamed arguments rarely occur in our data. Fig. 6.16 also indicates a rather large amount of functions with large  $DG$  sizes above 99. In some rare cases, the size is larger than 100 000. This aligns with our original motivation, that a manual inspection of the full space of possible solutions is either not practical or even not feasible in a reasonable amount of time. However, our search phase may collect object creators that would be discarded directly by a developer. Such cases artificially increase  $|DG|$ .

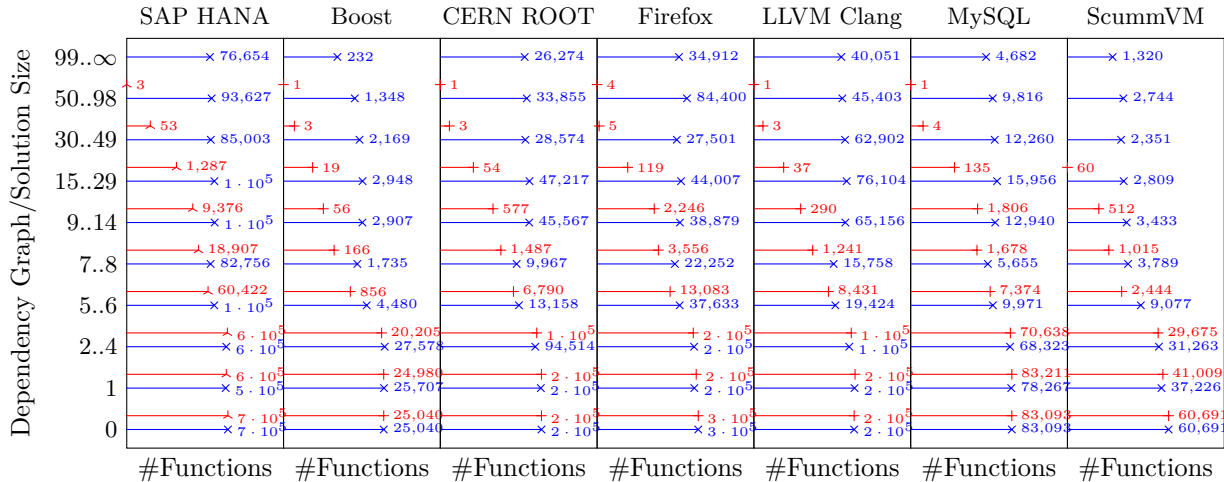


Figure 6.16: Dependency graph sizes — and solutions sizes — (x-axis in log-scale). E.g., Boost has 2169 functions with dependency graph sizes in range 30-49, and only 3 functions with solution sizes in this range.

We manually investigated unsolved functions. They contain in no particular order: a) templates, b) types defined but not implemented, c) types provided by the operating system d) types that were supposed to be non-constructable, and e) types we found no way to manually construct them. We expect that fine-tuning the system with domain knowledge could improve b), c) and d). A more sophisticated template analysis, which might be rather complex [252], may improve a).

— Answer RQ18 —  
 Our algorithm for object creation finds at least one solution to create all required objects for 94% to 99% of all functions in the evaluated projects.

## 6.2.6.4 RQ19 Verification of Solutions

RQ19 How does the quality of the solutions found by our approach compare to manually found solutions with respect to the criteria identified?

We compare object creations by developers found in the source code of SAP HANA versus those automatically proposed based on preferences stated by developers in our user study. The comparison involves manual tasks, therefore we evaluate only a subset of all object creations. To collect these examples, we extend our static analysis to report all locations in the source code where an object is created. We then filter them by a selection process and analyze the results. Additionally, we a) ask developers of SAP HANA to propose multiple problem instances and evaluate corresponding solutions, and b) manually investigate problem instances for the other projects.

**Selection of Examples:** We apply the following filtering steps:

1. SAP HANA consists of about 300 components that can be considered as own projects. We randomly select a set  $C_{50}$  of 50 components.
2. For each item in  $C_{50}$ , we collect the set  $OC$  of all source code locations where objects are created.
3. We only keep items in  $OC$  where all of the following conditions are true: file extension  $\in \{\text{cpp}, \text{cc}, \text{h}, \text{hpp}, \text{inl}, \text{inc}, \text{incl}, \text{hh}, \text{c}\}$ , source filename contains “[T/t]est” (test code), object not in namespace *testing* (test framework) nor *std* (standard library) nor  $X$  where  $X$  is an internal reimplementation of the standard library.
4. After filtering  $OC$ , we randomly select a set  $OC_{50}$  of up to 50 items.
5. We analyze each item in  $OC_{50}$ , and remove those that create artificial test objects such as a testfixture or a mock. This results in a final set of examples for each component.

Table 6.7 reports statistics for each step.

Step	Statistics After the Step
(0) Start	3 539 879 object creations
(1) 50 random components	#files: (2/1 311/267)
(2) Object creations	661 886 object creations
(3) Filter step	113 081 object creations per component: (0/17 086/2 262)
(4) 50 random examples	19 components with 0 examples
(5) Remove test objects	1 104 object creations per component: (0/49/22)

Table 6.7: Selectivity of the examples selection. Statistics annotated in (*min/max/mean*).

**Categorization:** For each example, we apply our approach and generate a solution. We compare this solution to the manual object creation indicated by the existing source code. Table 6.8 reports the results.

Out of 1 104 analyzed examples, the category *Shorter* contains 594 items, *Identical* 505 items, and *Longer* 5 items. Within *Shorter*, there are 113 cases where the solution used source code where the last change date is after the existing object creation, and in 481 cases before. Within *Identical*, there are 179 cases where the solution used source code where the last change date is after the existing object creation, and in 326 cases before.

Category	n	% $_{Total}$	$D_a$	%	$D_b$	%
Total	1 104	100.00			N/A	
Shorter	594	53.80	113	19.02	481	80.98
Identical	505	45.74	179	35.45	326	64.55
S + I	1 099	99.55	292	26.57	807	73.43
Longer	5	0.45			N/A	

Table 6.8: Solution sizes of  $ALG_o$  vs. manual solutions.

**Date Analysis:** Source code changes could impact the retrospective analysis. Our approach could propose to use code that was not available for a manual solution. Utilizing the version control system to use the specific version would require recompilations and static analyzes with an estimated effort of 138 d of execution time and 45 TiB of disk space. Due to limited resources, we instead extend our analysis to control for the described threat.

We use the version control system to calculate a date  $D_S$  for a solution, i.e., the last date when source code involved in a solution was modified, and identify the date  $D_M$  the manual solution was introduced. In Table 6.8,  $D_a$  reports number of cases for  $D_S > D_M$  and  $D_b$  for  $D_S \leq D_M$ .

**Manual Verification:** Developers of SAP HANA proposed 10 recent problem instances, i.e., functions they wanted to call. One function requires only fundamental arguments, the others require at least one object, i.e., they represent complex scenarios. We apply our approach and ask the developers to evaluate the results. In 8 cases, they determine the results are identical to their own solutions. In 1 case, the result is better due to the correct identification of a default argument in a header file. In 1 case, our approach found no result. However, the developers revealed they also found no solution for this case and a solution probably does not exist. Hence, our approach found identical or better solutions in all cases.

For each of the other projects, we randomly select 10 object types from files with paths containing the string  $[T/t]est$ . We apply our approach and report the results as a tuple (smaller/identical/larger) that shows our solution sizes in comparison to the solution sizes found in the source code. ScummVM, ROOT, and MySQL: all (0/10/0). Boost: (1/9/0). The smaller case involves a custom smart pointer that would require special case handling and domain knowledge of the project. Firefox: (1/9/0). Clang: (2/8/0). One smaller case is within a test framework not controlled by the project.

**Discussion:** In 45.74 % of all examples for SAP HANA, solutions found by our approach are identical to existing solutions and smaller in 53.80 %. The 5 larger cases involve complex template metaprogramming [138] and interfaces with a high number of implementations. Still, developers might choose other solutions due to specific requirements. However, in the context of test creation, the functionality for objects not under test is typically not important. Hence, we assume correctness of our results in such scenarios.

The date analysis indicates that the impact of code changes for the retrospective analysis is low. For the category *identical*, the source code of the calculated solution must have been available at the time the manual solution was introduced. However, in 35.45 % of all 505 cases,  $D_S < D_M$ . Hence, 36% is a threshold of expected cases. For the category *smaller*, 19.02 % is below this threshold, confirming the initial statement.

Given these results and considering the results of the manual verification, our approach is able to propose correct solutions to create objects.

— Answer RQ19 —

In 99.55 % of all 1 104 cases, our approach proposes solutions identical (45.74 %) or smaller (53.80 %) compared to existing solutions.

#### 6.2.6.5 RQ20 Comparison Against First-Working Approach

RQ20 How does our approach of a size-minimal solution compare against a first-working-solution approach?

**Methodology and Results:** Related work uses a first-working-solution approach, say  $ALG_{fw}$ . Here the function  $minSize$  (line 7 in Listing 6.1) is replaced by a check whether the subgraph is non-empty and if yes, the loop is aborted. We compare  $ALG_o$  against  $ALG_{fw}$  in terms of sizes of solutions for all functions. However, we only investigate functions that require additional objects of a type that is not provided by the function. These functions represent 38 % to 68 % of all functions depending on the project. We do not report the number of solutions for  $ALG_o$  and  $ALG_{fw}$ , as they are identical.

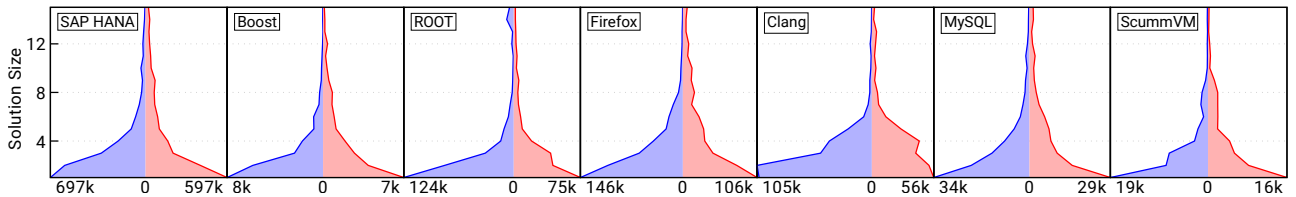


Figure 6.17: Comparisons  $ALG_o$  (left) versus  $ALG_{fw}$  (right) for functions. The histograms show how often each solution size occurred.

Fig. 6.17 shows the histograms of solution sizes for each project and approach. Fig. 6.18 presents only the cases where the solution sizes are not equal. Table 6.9 shows statistics for all cases. The execution time of  $ALG_{fw}$  is typically lower than  $ALG_o$ . However, the graph construction requires considerably more time compared to finding a solution that typically finishes within the fraction of a second. For example, the analysis for Clang shows the largest execution time with 0.15 s per function on average.  $ALG_{fw}$  may produce different results depending on the order of the input. Given the large number of functions, we expect that this randomness is not a threat, and we did not further investigate different orders.

Project	$ DG $ , all		$o \neq fw$ %	$ DG , \neq$	
	$o$	$fw$		$o$	$fw$
SAP HANA	3.10	6.81	42.07	4.45	13.26
Boost	2.57	3.90	31.80	3.13	7.31
ROOT	2.94	11.41	56.09	3.87	18.97
Firefox	2.88	5.94	44.69	3.67	10.51
Clang	2.62	11.64	60.17	2.96	17.94
MySQL	3.35	5.91	36.31	4.87	11.92
ScummVM	2.73	3.74	35.02	3.78	6.66

Table 6.9: Comparison between  $ALG_o$  and  $ALG_{fw}$ .

**Discussion:** Fig. 6.18 shows that size 1 does not occur for  $ALG_{fw}$ . This is expected because the design of  $ALG_o$  guarantees that  $|ALG_o(f)| \leq |ALG_{fw}(f)|$ . We removed all cases where  $|ALG_o(f)| = |ALG_{fw}(f)|$ . Hence,  $0 < |ALG_o(f)| < |ALG_{fw}(f)| \forall f$ .

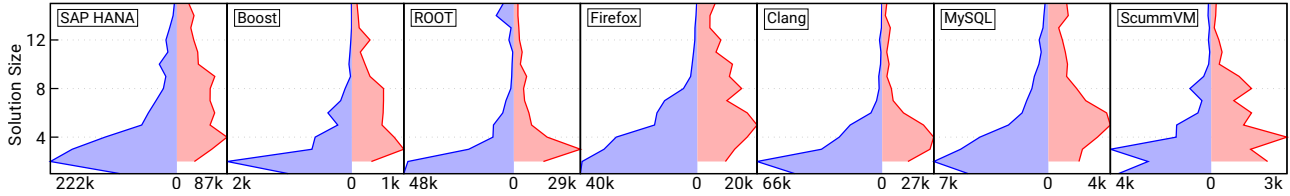


Figure 6.18: Presentation similar to Fig. 6.17. However, all cases where both solutions have equal sizes are ignored.

Over all projects, solutions based on  $ALG_o$  are on average by a factor 1.37 to 4.44 smaller for all functions or by a factor 1.76 to 6.07 smaller ignoring functions with identical solutions for both algorithms. Hence, in comparison to  $ALG_{fw}$ ,  $ALG_o$  can considerably reduce the amount of objects and therefore decrease the complexity of solutions. Fig. 6.18 also shows that  $ALG_o$  effectively reduces cases with large solutions. Therefore, we conclude that our approach improves  $ALG_{fw}$ .

— Answer RQ20 —

Solutions by  $ALG_o$  require up to 6 times less objects on average compared to a first-working-solution approach.

### 6.2.7 Threats to Validity

We discuss several threats to validity that we identified for our work.

#### 6.2.7.1 User Study

Participants in the user study may not have the professional experience to answer the questions [40]. We reduce this threat by sending the survey to professional developers. However, we are unaware of the number of participants without C++ experience. All participants are related to our industrial partner. However, we are not aware of any company policy that may influence our anonymous survey. With respect to diversity, the recipients are distributed worldwide and have different professional experience.

The user study might be ambiguous or the lists of items might be incomplete. We reduce this threat by a trial run.

#### 6.2.7.2 Reliability

We collected a set of 7 projects for our evaluation. However, the composition did not follow a reproducible methodology, because we are unaware of a definitive list of large C++ projects. Due to the regulations of our industry partner, the implementation of our approach is not publicly available and an exact reimplementaion of our approach may not be feasible. We carefully tested our implementation with an extensive test suite of collected C++ code examples and therefore expect that the conclusions are reproducible.

### 6.2.7.3 Construct Validity

Our evaluation contains a retrospective analysis. We are unaware of the reasons why a specific object creation option was selected in the past or whether such preferences have changed over time. A/B testing could mitigate this threat. However, it would require extensive resources to do such testing in large scale, therefore it was out of scope for our work.

### 6.2.7.4 Internal Validity

The search phase may find creators that are technically feasible, but practically not. Also, we may iterate uninteresting creators for the graph traversal. Thus, we may investigate cases that would be discharged directly by developers. This may produce more work for  $ALG_o$  but does not affect our conclusions.

### 6.2.7.5 External Validity

We assume that our approach can be generalized to other object-oriented programming languages with type information. However, in small projects, object creation may not be noticed as a problem. The user study results for time efforts may be specific to the rather strict C++ type system. The ranking of criteria provides guidance for other programming languages, too.

## 6.2.8 Related Work

Several other work on testing object-oriented programs either focus on the broader problem to generate a desirable object state [92, 142, 182, 209, 247, 249, 255, 268] or do not consider the object creation and, for example, capture objects during runtime [142]. Our work focuses on the specific part of object creation and does not aim to generate a desirable state.

The work of Thummalapenta et al. [247] is closely related to our work. They also identify the problem of object creation as challenging and propose to use a keyword-based code search in method bodies within an intra-procedural analysis. However, they state that such analysis “is less precise than inter-procedural analysis” and only used due to scalability reasons. Due to the keyword-based search, their approach does not require or use type information. The type detection by an inter-procedural analysis of our approach provides accurate results and is still fast.

Several related work provide approaches for test generation in object-oriented programming languages [13, 92, 142, 209, 249]. Such approaches require a mechanism for object creation. They either (a) search for a constructor or generate the object with a general mechanism provided by the programming language [13, 142] or (b) recursively traverse the required dependencies for object creation until they find the first working solution [92, 249]. However, considering only the first working solution is undesirable in real-world projects according to the results of our survey. Cseppentő and Micskei provide further evaluation of test generation tools and their support regarding objects [58].

Previous work, where the evaluation targets comparatively small projects, recognize the challenges for the creation of complex objects [12, 228, 249].

The tools KLOVER [177, 258] and FSX [259, 260] automatically generate unit tests for large C++ projects. The examples shown for FSX contain a default constructor call, hence object creation is supported to some degree. However, the exact support remains unclear. Garg et al. target unit test generation in C++ with directed random test generation [98].

We are not aware of other studies on developers' preferences for the optimization version of the object creation problem. We are unaware of existing tools for C++ or Java that respect such developer preferences.

### 6.2.9 Conclusions

The task of object creation in large C++ projects can be a time-consuming challenge for developers. Our approach automatically finds options for the object creation problem in more than 94% of all cases and solutions to create all required objects for 94% to 99% of all functions. Therefore, our approach can provide significant time reductions for developers. In addition, our approach can find solutions that better align with preferences of developers compared to solutions found manually. Thus, using our approach can improve code quality. Finally, solutions found by our approach better align with preferences of developers compared to solutions found by a random approach that is used in related work.

In practice, developers could consider additional requirements for creating objects. Or even more, they might add new constructors to solve the problem. However, even in such cases, our approach can provide a list of alternative options and additional insights for *missing* options.

While our work focuses on large C++ projects, we expect that the results generalize to other object-oriented programming languages with type information. We also expect that the results of our work allow other researchers to propose techniques with higher practical acceptance.

Future work on the connection between the creation of objects and their set-up into a desired state may provide additional benefits for automated tools and the manual work of practitioners.

## 6.3 Mock Proposal

A significant share of the effort to create unit tests in object-oriented software can be attributed to constructing a desired object state. This problem typically includes object creation and object setup, two tasks that can be complex in large projects and in the case of interactions with the global state. Therefore, developers may decide to *mock an object*, i.e., they simulate the behavior of objects required for a test with simpler implementations<sup>1</sup>. However, they have to decide which objects should be mocked.

To support developers with writing unit tests, we address the problem of selecting objects to be mocked and propose heuristics for automated mock recommendations. We complement it by an algorithm that minimizes the total number of required mocks for simultaneous instantiation of multiple objects. Our core methods are graph analysis and traversal algorithms.

An evaluation on seven large C++ projects shows that our algorithm can reduce the total amount of mocks if multiple objects are instantiated.

Mock an object

<sup>1</sup> Note that we use the term mock and do not differentiate between mocks, fakes, stubs, dummies or other specialized concepts [234].

### 6.3.1 Introduction

Programming languages that allow to use and instantiate *objects* are widely used today [41, 52, 100]. Developing unit tests in such cases requires the construction of object instances in order to test their state, methods, and interactions. Hence, a test setup must create such objects and set-up their desired states [12]. A wide range of research work proposed multiple techniques to automatically create a desired state for an object [92, 182, 209, 249, 255, 268]. Due to visibility and dependency issues, this can be a challenging task [142].

Frequently for testing purposes, developers do not want to instantiate and set-up a target object. An object might contain unintended functionality or require expensive resources. In such cases, they substitute the regular object by creating a *mock object*. Such a mock mimics a specific behavior required for a test while avoiding the unwanted characteristics of the regular object. However, mocks have also disadvantages, such as the effort of their creation, or possible deviations from the original functionality [234]. In the face of these trade-offs, it might be challenging for developers to decide which objects should be mocked, if any (the *mock selection problem*).

Mock object

Consequently, developers could benefit from approaches that automatically propose mock recommendations. Our contributions are as follows:

- We propose approaches for recommending which objects to mock. In addition to two heuristics inspired by common practices, we propose an algorithm that reduces the overall number of mocks if multiple types or functions are tested simultaneously.
- An evaluation of our approach on seven large C++ projects.

### 6.3.2 Motivation

We provide examples for mock selection to illustrate the involved issues.

#### 6.3.2.1 Mocking for dependency reduction

Fig. 6.19 shows an example with multiple classes. Assume that one wants to instantiate the class `Person`. Such an object requires an object of type `Office` which in turn requires an object of type `Building`. The constructor for `Building` requires multiple other objects that are unclear to create.

```

1 struct Building {
2     Building(Description& d, Address& a, Dimension& d, Time& t,
3         PowerMap& p) : /* setup of members */ {}
4     /* several members and complex function */
5 };
6 struct Office {
7     string name;
8     Office(string id, Building& b) : name(b+id) {}
9 };
10 struct Person {
11     Person(Office& office) : office(office) {}
12     private:
13         Office& office;
14     /* functions */
15 };

```

Figure 6.19: Mock object for dependency reduction.



In such a situation, a developer could create a mock object for `Building` to imitate it because it is not related to `Person` and is costly to create.

`Office` is not a good candidate for mocking, as the class stores only a value and requires only an object of type `Building`.

We conclude that this example highlights two mocking goals:

1. Reduce complexity or the number of dependencies not relevant for a test.
2. Avoid mocking of so-called *value classes*, i.e., classes that only store values and have no additional functionality.

### 6.3.2.2 Multiple mocking options

Objects can require multiple tests for multiple functions and interactions. In such situations, a developer has to decide multiple times whether an object of a certain type should be mocked. The developer might decide against a mock the first time but would decide in favor of mocking if the same object type is encountered several times. Therefore, it can be beneficial to create a mock if this mock will be used by multiple functions. However, such an information might not be directly available to developers in large projects. In practice, objects often depend on other objects that must be created first. The amount of such dependent objects can reach rather large numbers.

```

1 struct X {
2     int a;
3     X(string& s) : a(s.length()) {}
4 };
5 /* Y,M,N,O,P,Q,R all defined similarly to X */
6
7 struct R {int a; R(string& s):a(s.length()){}};
8
9 struct A {int a; A(X& x, Y& y) : a(x.a + y.a){}
10    A(X& x, Y& y, M& m) : a(x.a + y.a + m.a){}};
11
12 struct B {int a; B(M& M, N& N) : a(M.a + N.a){}
13    B(Y& Y, M& M, N& N) : a(Y.a + M.a + N.a){}};
14
15 struct C {int a; C(O& O, P& P) : a(O.a + P.a){}
16    C(X& X, M& M, N& N) : a(X.a + M.a + N.a){}};
17
18 struct D {int a; D(Q& Q, R& R) : a(Q.a + R.a){}
19    D(X& X, Y& Y, N& N) : a(X.a + Y.a + N.a){}};
20
21 struct E {int a; E(Q& Q, R& R) : a(Q.a + R.a){}
22    E(X& X, Y& Y, N& N) : a(X.a + Y.a + N.a){}};
23
24 struct ClassUnderTest{
25     int function1(A& a) {return a.a;}
26     int function2(B& b) {return b.a;}
27     int function3(C& c) {return c.a;}
28     int function4(D& d) {return d.a;}
29     int function5(E& e) {return e.a;}
30 };

```

Figure 6.20: Multiple possible mocking options.

Finding a *configuration with a minimal number of mock objects* can be a rather complex task in such situations. The example shown by Fig. 6.20 is rather complex but reflects practical problems in large software projects. The goal is to test the class `ClassUnderTest`. A developer could inspect each function and decide to create 5 mock objects: `A`, `B`, `C`, `D`, `E`. This

Configuration with a minimal number of mock objects

might seem reasonable because they appear as the first level of indirection. However, this approach would not result in a minimum amount of mocks for our example. A developer could apply a more elaborated approach that skips the first level of dependencies and selects the minimum from the second level for each function. In this variant, 8 mock objects would be required:  $X, Y, M, N, O, P, Q(2), R(2)$  where the number indicates how often a mock is used if used multiple times. This solution would require even more mock objects than the first variant. However, the minimal solution requires only 4 mock objects (e.g.,  $X(4), Y(4), M(3), N(4)$ ). Therefore, it can be beneficial to consider a larger scope for automated recommendations.

### 6.3.3 Approach

Automated mock recommendations, i.e., recommendations where a mock object should be introduced to imitate the behavior of the original object, depend on the goal of the imitation. Therefore, we study several heuristics to reflect these goals that can be used separately or combined.

#### 6.3.3.1 Requirements

A requirement for mock recommendation is a precise understanding of the source code. Additionally, we need to understand object type dependencies, i.e., we need to know if a target object requires additional objects for its creation. The Clang plug introduced in Section 6.2.4.2 provides a precise understanding of the source code and the object dependency graph introduced in Section 6.2.4.1 contains all required information on object dependencies.

#### 6.3.3.2 External Dependencies

The filesystem provides a heuristic of whether a dependency is external or internal. Related functionality is typically grouped within the same directory. Therefore, we can expect that all objects defined by source code within the same directory are related. In contrast, objects with source code in other directories are classified as external and it is recommended to create a mock object for such external objects. We use the identifier  $MD$  for this heuristic.

Functionality grouped by directories

#### 6.3.3.3 Value of a Class

We define a *value class* as a class that acts only as a data container and contains only a small amount or no logic. We expect them to be simple to instantiate without any requirements. We argue that it is generally not valuable to mock such classes. A mock object would most likely only reimplement the value class and is therefore redundant to the original object.

The detection of value classes requires a heuristic. We propose to use the type of members and the types of parameters for the constructors as indicators. For this purpose, we define that a type is simple if:

1. The type is a fundamental type ([138], 6.9), such as `bool`, `int`, or `float`.
2. The type, as represented by a class, has an implicit or explicit default constructor that is public and not deleted ([138], Clause 15), i.e., we can call a constructor with empty arguments to create an object of this type.

3. After removing all types listed in a manually pre-defined whitelist, the type falls in any of the previous categories. The whitelist contains project specific types such as a string class, types that allow allocation and deallocation of memory or types that provide logging functionality.

Therefore, we classify an object as a value class and do not recommend mocking if all parameters and members of an object are simple, i.e., it is possible to create or call them without additional requirements. Otherwise, we recommend mocking. We use the identifier *MV* for this heuristic.

### 6.3.4 Algorithm for Minimal Amount of Mocks

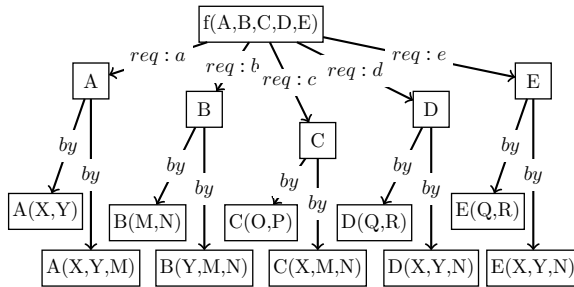


Figure 6.21: Simplified dependency graph for Fig. 6.20.

As discussed in Section 6.3.2.2, the analysis of multiple functions can lead to different mocking decisions compared to the case of a single function. We propose to consider such cases of multiple functions in mocking recommendations. In our approach, we combine the dependency graphs of multiple functions to a single dependency graph and implement an algorithm to find a recommendation with a globally minimal number of mocks.

To combine the dependency graphs for multiple functions  $f_1, \dots, f_n = S_f$ , we then apply the following steps:

1. Create the set  $S_t$  of all parameter types of each function in  $S_f$ .
2. Introduce a new artificial function  $f$  in such a way that the parameter list of  $f$  contains exactly all elements of  $S_t$ . For this artificial function  $f$ , we create the dependency graph  $G_d$  as described in Section 6.2.4.3.
3. Call  $G_d$  the combined dependency graph for all elements in  $S_f$ . Fig. 6.21 presents the combined dependency graph for Fig. 6.20.

We can now use the combined dependency graph for our analysis of mock objects for multiple functions. A *trivial approach* would recommend mock objects for all parameter types of the root function. Note that the result of this trivial approach would be identical to the result of a single function analysis. Such an approach does not always create an optimal solution as shown by Fig. 6.20. The trivial approach would recommend 5 mock objects, while 4 mock objects would be enough.

Trivial approach

In contrast to the trivial approach, there is the *full enumeration approach* where we iterate over all possible mocking recommendations for all types in the combined dependency graph and select from all valid solutions the solution with the minimal number of mock objects. This guarantees to find the global minimum but is in practice not feasible due to the possibly large number of  $2^n$  configurations for  $n$  different types in the dependency graph.

Full enumeration approach

Instead of a full enumeration approach, we propose in Fig. 6.22 an algorithm  $ALG_{mm}$  that exploits our graph data structure and properties of the dependency graph to avoid the iteration over all  $2^n$  types as for the full enumeration approach. The key idea of our algorithm originates from the observation that edges in our dependency graph are directed. For Fig. 6.20, where we must resolve the dependency of type  $A$ , we do not have to consider all possible mock configurations for the subgraph of  $A$ . For example, the option to propose a mock object for type  $A$  and types  $X, Y$  must not be evaluated, because objects of type  $X$  and  $Y$  are not required if we introduce a mock object for type  $A$ . In terms of our dependency graph, we can abort the traversal of a subgraph below a type node if the type should be mocked.

Further, in these cases where we have several options to create an object, we do not have to consider all combinations of all options. In the case of our example in Fig. 6.20, we do not have to inspect the configuration to mock  $Q, R$  and  $X, Y, N$  below type  $E$ . It is enough to select one of the possible configurations to create an object.

Our own algorithm

```

1 alg_mm(DependencyGraph graph) -> Set(Type) {
2   return getMinimumMocking(graph.rootnode)
3 }
4 getMinimumMocking(Node node) -> Set(Type) {
5   Set(Set(Type)) allValidMocks = allValidMocks(node)
6   return minimum(allValidMocks) //suitably defined
7 }
8 allValidMocks(Node node) -> Set(Set(Type)) {
9   if (node is empty) return EmptySet
10  Set(Set(Type)) solutions
11  if (node is not ROOT) add node.type to solutions
12  List(Set(Set(Type))) argSolutions
13  for each argument in node
14    for each Set(Node) creatorNodes in argument
15      Set(Set(Type)) argSolution
16      for each Node node in creatorNodes)
17        // each option for an argument
18        Set(Set(Type)) subgraph = allValidMocks(node)
19        if (subgraph is not empty)
20          add each element of subgraph to argSolution
21        if (argSolution is not empty)
22          add argSolution to argSolutions
23  if (argSolutions is not empty)
24    Set(Set(Type)) combined = combine(argSolutions)
25    add each element of combined to solutions
26  return solutions
27 }

```

Figure 6.22: The algorithm  $ALG_{mm}$  to find recommendations for mocking that minimize the total amount of mocks. Fig. 6.23 presents the combine function.

$ALG_{mm}$  avoids the enumeration of  $2^n$  configuration for  $n$  different types within the dependency graph. However,  $ALG_{mm}$  includes several parts that can have a superlinear evaluation complexity that depends on the number of types and function parameters. It is trivial to parallelize the implementation of  $ALG_{mm}$ , because the subgraphs allow independent processing without any write conflicts. However, in such cases where the execution time of  $ALG_{mm}$  would be too large, we propose a heuristic variant  $ALG_{mm-heuristic}$  that modifies  $ALG_{mm}$  in two aspects. These modifications reduce the time requirements but could result in a suboptimal solution. The implementation of  $ALG_{mm-heuristic}$  follows Fig. 6.22. We omit to show the full implementation and we only describe the two modifications:

```

1 /** Combine all possibilities for each argument. */
2 combine(<...> solutions) -> Set(Set(Type)) {
3   Set(Set(Type)) result
4   if (solutions is empty)
5     add empty set to result
6     return result
7   Set(Set(Type)) first = solutions[0]
8   Set(Set(Type)) rest = solutions[1..] //all but first
9   Set(Set(Type)) restCombined = combine(rest)
10  for each Set(Type) firstItem in first)
11    for each Set(Type) restItem in restCombined)
12      joined = firstItem union restitem
13      add joined to result
14  return result
15 }

```

Figure 6.23: Helper function for algorithm  $ALG_{mm}$  to create all combinations.

1. For each call of `allValidMocks`, decide whether the result should contain the type of the current node or if the subgraph is further investigated.
2. For each call of `combine`, decide whether the result should only contain the union of all types or all possible combinations of all types.

In our experiments, heuristics based on the depth of the graph, or the number of children, or the current amount of possible solutions lead to practicable results in terms of execution time and quality of the solution.

### 6.3.5 Evaluation

We investigate the following research questions (RQ):

- RQ21 How does the heuristics  $MD$  and  $MV$  affect the number of required objects for function calls?
- RQ22 Can mocking recommendations that simultaneously consider many target functions ( $ALG_{mm}$ ) reduce the number of mock objects compared to mocking recommendations that consider the target functions one by one?

We investigate functions as they are typically targeted by unit tests. We study the same projects as in Section 6.2. Table 6.10 repeats their characteristics, further details are provided by Section 6.2.

Project	#Obj. types	#Functions	SLOC
SAP HANA 2018-11-24	735 194	4 210 541	11 065 382
Boost 1.66 [65]	30 548	53 528	4 392 925
CERN ROOT 6.13/08 [66]	100 705	730 507	3 417 362
Firefox 55.0.3 [68]	134 554	940 346	7 343 242
LLVM Clang 6 [69]	88 913	591 112	242 032
MySQL 8.0.11 [70]	54 360	199 692	3 791 989
ScummVM 2.0.0 [72]	13 527	148 350	1 830 628

Table 6.10: List of evaluation projects.

We use a workstation with 4 CPU, 160 cores with 2.10 GHz, and 1 TiB RAM. In the cases when the execution time of algorithm  $ALG_{mm}$  exceeds 30 min or the memory consumption exceeds the available hardware, we continue with a heuristic version as described in Section 6.3.3.

As discussed in Section 6.3.3.1, we first have to create dependency graphs of argument types. To collect all required objects, we have two alternative approaches: We can collect all distinct object types or functions.

We use our Clang plugin (Section 6.2.4.2) for static analysis to generate a list  $F$  of all functions and information about their argument types. For each function  $f$  in  $F$ , we create a dependency graph  $DG_f$  as described in Section 6.2.4.3. The root node  $n_r$  of  $DG$  represents  $f$ , the child nodes on the first level  $n_1$  connected to  $n_r$  represent the arguments of  $f$ . Each node in  $n_1$  then represents the dependency graph for the corresponding parameter type. We finally obtain the set  $S_{DG}$  of all dependency graphs for all functions. For each element in  $S_{DG}$ , we apply both heuristics as described in Section 6.3.3.

Fig. 6.24 shows the results for  $MD$  and Fig. 6.25 shows the result for  $MV$ . Table 6.11 presents the results aggregated as averages (arithmetic mean). The table shows the percentage of solved functions, i.e., functions where we found an option to create them. We can apply our approach only for such cases. Then,  $|DG|$  represents the size of a dependency graph before finding a solution. Next,  $|S|$  represents the size of the (minimal) solution. Finally,  $|MD|$  and  $|MV|$  represent the sizes of dependency graphs after we introduce mocks according to the heuristics based on directories and values, respectively. For comparison, we also provide the average number of arguments per function,  $|args|$ .

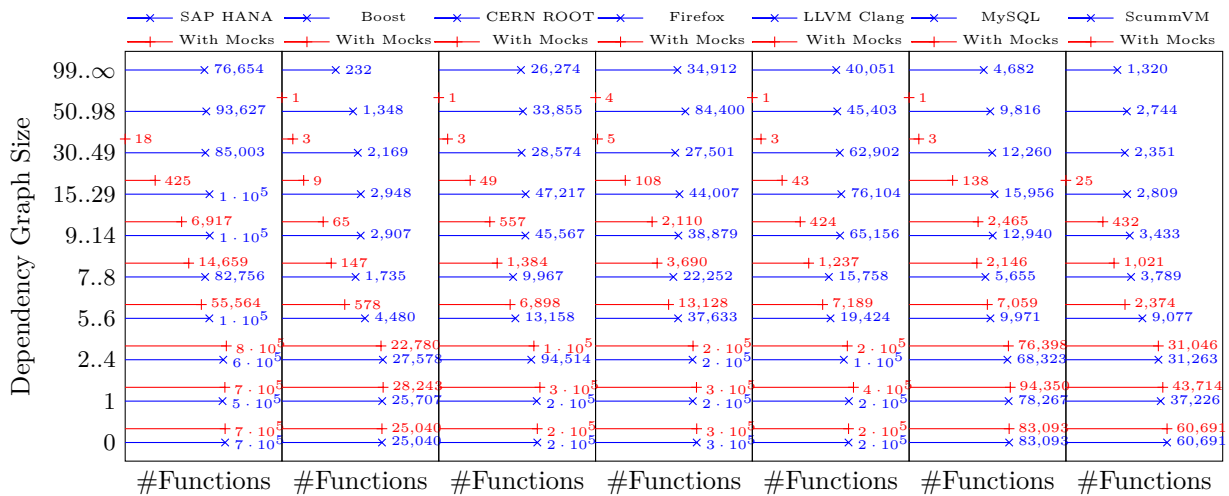


Figure 6.24: Dependency graph sizes before — and after — mock object recommendation based on directory. Log-scale x-axis.

Fig. 6.24 and Fig. 6.25 indicate that simple heuristics for mock object recommendation can significantly reduce the size of dependency graphs. On average, as shown by Table 6.11, the number of required objects is a factor 1.5 smaller if we follow the recommendations of  $MD$ .

Interestingly,  $|MV| < |args|$ . This seems to be a contradiction because each argument requires at least one object. However, unnamed and default arguments do not require objects. The heuristic  $MV$  recommends mocking all complex types, i.e., typically these types that depend on additional types. In contrast, simple types typically do not depend on additional objects. Therefore,  $MV$  often shows solutions with size one and  $|MV| = |args|$  in

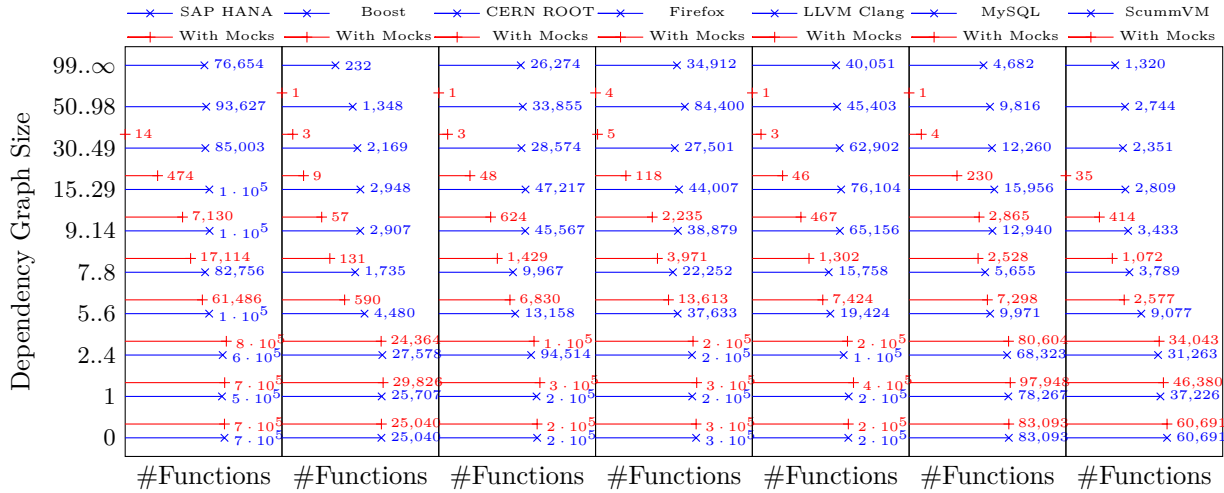


Figure 6.25: Dependency graph sizes before and after mock object recommendation based on mock value. Log-scale x-axis.

such cases. However, unnamed arguments and default arguments do not require additional object creation. Therefore, such arguments are “ignored” and, consequently,  $|MV| < |args|$ . For practical purposes,  $MV$  might be only useful for “negative” recommendation, i.e., to filter all types where mocking is not recommended. It may not be advisable to mock all other objects. The heuristic seems to be effective for such negative cases.

Project	%Solved	$ DG $	$ S $	$ MD $	$ MV $	$ args $
SAP HANA	97.98	11.36	2.20	1.76	1.61	1.63
Boost	98.31	4.56	1.78	1.50	1.43	1.47
CERN ROOT	96.11	14.49	1.85	1.20	1.15	1.18
Firefox	97.16	15.16	1.95	1.47	1.37	1.39
LLVM Clang	94.74	18.96	2.03	1.32	1.26	1.28
MySQL	98.38	9.10	2.24	1.66	1.56	1.59
ScummVM	99.89	5.19	1.56	1.24	1.17	1.17

Table 6.11: Average number of objects required to create per category.

Answer RQ21

The mock heuristic  $MD$  can considerably reduce the amount of required objects for function calls by a factor of 1.5 on average. The heuristic  $MV$  can correctly identify objects where mocking is not beneficial.

Next, we evaluate the algorithm  $ALG_{mm}$  for minimal amount of mocks.  $ALG_{mm}$  targets functions, therefore we change the evaluation focus to multiple functions. First, we group all functions by source file. This reflects typical developer practice for test organization, that is, the test code follows the same file system organization as the code under test. Then, for each group, we introduce a new artificial function  $f$  that contains the parameters of all functions in this group. We have two choices to select the parameters for  $f$ . We can create the union of all parameters as a set where each parameter type is distinct (*Distinct*), or we can append all parameters as a list (*All*). We also compare against the baseline of single functions (*Single*).

We apply our approach for all three variants. We calculate the sum  $\#M$  of recommended mocks, the sum  $\#P$  of parameters found in functions and their ratio as percentage  $\#M/\#P\%$ . Table 6.12 shows the results.

Project	Single			Distinct			All		
	#M	#P	$\frac{\#M}{\#P}\%$	#M	#P	$\frac{\#M}{\#P}\%$	#M	#P	$\frac{\#M}{\#P}\%$
SAP HANA	190 765	3 958 115	4.82	90 537	527 617	17.16	67 151	4 064 486	1.65
Boost	7 564	133 864	5.65	4 669	26 530	17.60	3 714	137 998	2.69
ROOT	38 898	783 146	4.97	21 342	135 741	15.72	13 988	804 508	1.74
Firefox	111 598	1 313 459	8.50	78 480	286 043	27.44	54 391	1 336 700	4.07
Clang	57 805	1 022 226	5.65	28 837	166 472	17.32	16 778	1 048 926	1.60
MySQL	20 136	465 061	4.33	16 414	80 137	20.48	10 929	479 828	2.28
ScummVM	11 675	178 190	6.55	11 497	36 786	31.25	11 381	180 406	6.31

Table 6.12: Results for  $ALG_{mm}$ .

Table 6.12 shows the benefits of considering a larger scope. The number of recommended mocks ( $\#M$ ) is lower by a factor of 1 to 3 for a file scope compared to single functions. The ratios  $\#M/\#P$  differ by a factor of 5 to 10 for *Distinct* and *All*. Our algorithm successfully detects types that occur multiple times and prioritizes mock recommendations for such types.

It seems unexpected that *Distinct* and *All* must analyze more types than to *Single*. This effect is caused by default and unnamed arguments. They can be exploited by the *Single* but are not available for *Distinct* and *All*.

For ScummVM, the differences for  $\#M$  is small. The dependency graphs for ScummVM are rather small resulting a low amount of mock configurations. Hence, mocks do not reduce the the number of required objects.

— Answer RQ22

A broader scope reduces the amount of required mock objects by a factor of 1 to 3 compared to considering single function for finding mock candidates.

### 6.3.6 Conclusions

We studied multiple heuristics for mock recommendation and proposed a method for minimal mock recommendations. Our evaluations show the possible practical benefit of these approaches. Future work is required for analyzing the quality of the mocking recommendations.

## 6.4 Summary

We proposed a technique that allows dynamic unit test extraction which is practical for large software projects. We applied this technique to SAP HANA and extracted several hundred unit tests that were all accepted by developers in code reviews, showing the practical usefulness of our technique. In addition, we also proposed several other techniques to help developers create unit tests, namely an automated approach to determine object creation steps and automated mock recommendations. Evaluations for both approaches show their benefits for large projects.

By utilizing a multi-stage testing strategy, our approach can considerably reduce test costs while maintaining the same overall degree of quality for the software under test. Hence, we consider the approach practical.



# 7 | Conclusions

As we introduced in Chapter 1, large software projects require quality assurance and testing but the corresponding costs for such activities can be substantial. Therefore, it is important to reduce such test costs. In our work, we analyzed root causes for test costs in large projects and we have discussed in Chapter 2 and shown in Chapter 5 that the size of large software project results in specific issues that are not significant for projects of smaller size.

We proposed and evaluated a general approach for test cost reduction in Chapter 5 that tackles the superlinear increase in test executions over time. In Chapter 6, we tackled the problem of system tests with large execution times by proposing to dynamically extract unit tests. Our approach extends approaches proposed by related work by adapting them for large projects and increasing their practical usefulness. In addition, we proposed several approaches to reduce the effort for writing unit tests and therefore motivating developers to create more unit tests instead of system tests.

Our evaluations show that our proposed approaches decrease test costs by a substantial amount and reduce the negative effects of tests with large execution times without loss in quality. We also believe that our approaches can be adapted to other projects, even with different programming languages and environments. However, they might only be interesting if test costs and test execution times are significant. For small projects with negligible test costs and test suites with less than 30s execution time, the effort to implement our approaches may not be justified.

In addition to our core contributions, we also investigated several questions related to coverage data. We build on related work to design efficient algorithms and data structures for the analysis of coverage data and we extend the existing knowledge about the relationship between coverage and faults by two empirical studies. We do not claim to provide a definitive answer to whether coverage data provides any additional information on faults. However, our studies suggest that coverage can be useful in estimating and directing testing activities. Future work is required to extend the state of knowledge for this topic and to investigate possible confounding variables and alternative explanations.

There are also several areas for future work. The process of generating coverage for large projects can be further investigated. Which coverage variant should be targeted and how can it efficiently be measured? Would it be beneficial to have a mix of different variants and how can we avoid issues such as random coverage? The testing process itself of large projects contains several further challenges due to the size of these projects. How

to solve the problem of flaky tests? How to select tests for different testing stages? We also see several possible future work for our proposed approaches. Our implementation for dynamic unit test extraction has several limitations. Some of them require further implementations such as support for more C++ functionality, but there are also conceptual limitations such as the challenge to extract unit tests from system tests if system tests read and write a large state but unit tests should be small, readable and maintainable.

We conclude that large projects provide very interesting opportunities for research. Due to the size of such projects, they show issues that may not be noted for smaller projects. These unsolved issues provide new insights for the understanding of software engineering. This is also reflected by the research community as the number of publications related to specific issues such as flakiness or very large test suites considerably grow in numbers since 2010. We hope that we can motivate further research with the work done for this thesis.

# 8 | Bibliography

- [1] Advanced Micro Devices (AMD). 2017. *AMD64 Architecture Programmer's Manual*. Technical Report. AMD. AMD number: 24592. (Cited on page: 23)
- [2] Advanced Micro Devices (AMD). 2019. *Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors*. Technical Report. AMD. AMD number: 56305. (Cited on page: 23)
- [3] Iftekhhar Ahmed, Rahul Gopinath, Caius Brindescu, Alex Groce, and Carlos Jensen. 2016. Can Test-  
edness Be Effectively Measured? In *Proceedings of the 24th ACM SIGSOFT International Symposium  
on Foundations of Software Engineering* (Seattle, WA, USA) (*FSE 2016*). Association for Computing  
Machinery, New York, NY, USA, 547–558. ISBN 9781450342186 DOI [10.1145/2950290.2950324](https://doi.org/10.1145/2950290.2950324) (Cited  
on pages: 55, 58, 59, 60, 65, 68, 70, 77, 88, 90, 91, and 92)
- [4] Amogh Akshintala, Bhushan Jain, Chia-Che Tsai, Michael Ferdman, and Donald E. Porter. 2019. x86-64  
Instruction Usage Among C/C++ Applications. In *Proceedings of the 12th ACM International Conference  
on Systems and Storage* (Haifa, Israel) (*SYSTOR 2019*). ACM, New York, NY, USA, 68–79. ISBN  
9781450367493 DOI [10.1145/3319647.3325833](https://doi.org/10.1145/3319647.3325833) (Cited on pages: 30, 123, and 125)
- [5] Sara Alspaugh, Kristen R. Walcott, Michael Belanich, Gregory M. Kapfhammer, and Mary Lou Soffa.  
2007. Efficient Time-Aware Prioritization with Knapsack Solvers. In *Proceedings of the 1st ACM Inter-  
national Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held  
in Conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering  
(ASE) 2007* (Atlanta, Georgia) (*WEASEL Tech 2007*). Association for Computing Machinery, New York,  
NY, USA, 13–18. ISBN 9781595938800 DOI [10.1145/1353673.1353676](https://doi.org/10.1145/1353673.1353676) (Cited on pages: 107 and 120)
- [6] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing* (second ed.). Cambridge University  
Press, New York, NY, USA. ISBN 1107172012, 9781107172012 (Cited on page: 23)
- [7] Sundaram Ananthanarayanan, Masoud Saeida Ardekani, Denis Haenikel, Balaji Varadarajan, Simon  
Soriano, Dhaval Patel, and Ali-Reza Adl-Tabatabai. 2019. Keeping Master Green at Scale. In *Proceedings  
of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) (*EuroSys 2019*). ACM, New York, NY,  
USA, 1–15. ISBN 9781450362818 DOI [10.1145/3302424.3303970](https://doi.org/10.1145/3302424.3303970) (Cited on page: 20)
- [8] James Andrews, Lionel Claude Briand, and Yvan Labiche. 2005. Is Mutation an Appropriate Tool for  
Testing Experiments? In *Proceedings of the 27th International Conference on Software Engineering* (St.  
Louis, MO, USA) (*ICSE '05*). Association for Computing Machinery, New York, NY, USA, 402–411.  
ISBN 1581139632 DOI [10.1109/ICSE.2005.1553583](https://doi.org/10.1109/ICSE.2005.1553583) (Cited on page: 92)
- [9] Artur Andrzejak and Thomas Bach. 2018. Practical Amplification of Condition/Decision Test Coverage  
by Combinatorial Testing. In *2018 IEEE International Conference on Software Testing, Verification and  
Validation Workshops (ICSTW 2018)*. IEEE Computer Society, Washington, DC, USA, 341–347. DOI  
[10.1109/ICSTW.2018.00070](https://doi.org/10.1109/ICSTW.2018.00070) (Cited on page: 6)
- [10] ANSI. 1986. *American National Standard for Information Systems – Coded Character Sets – 7-Bit  
American Standard Code for Information Interchange (7-Bit ASCII)*. Technical Report ANSI X3.4-1986.  
American National Standards Institute. (Cited on page: 24)
- [11] Howard Anton and Anton Kaul. 2019. *Elementary Linear Algebra* (twelfth ed.). John Wiley and Sons,  
USA. ISBN 9781119268048, 9781119406778 (Cited on page: 34)
- [12] Andrea Arcuri, Gordon Fraser, and René Just. 2017. Private API Access and Functional Mocking in  
Automated Unit Test Generation. In *2017 IEEE International Conference on Software Testing, Verification  
and Validation (ICST)*. IEEE Press, Piscataway, NJ, USA, 126–137. DOI [10.1109/ICST.2017.19](https://doi.org/10.1109/ICST.2017.19) (Cited  
on pages: 180 and 182)
- [13] Shay Artzi, Michael Dean Ernst, Adam Kiezun, Carlos Pacheco, and Jeff H. Perkins. 2006. Finding the  
Needles in the Haystack: GEnErating Legal Test Inputs for Object-Oriented Programs. In *M-TOOS: 1st*

- Workshop on Model-Based Testing and Object-Oriented Systems*. M-TOOS, Portland, OR, USA, 27–34. (Cited on page: 180)
- [14] Git authors. 2020. *Git Distributed Version Control System*. Retrieved 2020-01-10, archived by Internet Archive at <https://web.archive.org/web/20200110161018/https://git-scm.com/> from <https://git-scm.com/> (Cited on pages: 62 and 75)
- [15] LCOV authors. 2020. *LCOV - The LTP GCOV Extension*. Retrieved 2020-01-10, archived by Internet Archive at <https://web.archive.org/web/20200110161124/http://ltp.sourceforge.net/coverage/lcov.php> from <http://ltp.sourceforge.net/coverage/lcov.php> (Cited on pages: 25 and 30)
- [16] Thomas Bach, Artur Andrzejak, and Ralf Pannemans. 2017. Coverage-Based Reduction of Test Execution Time: Lessons from a Very Large Industrial Project. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2017)*. IEEE Computer Society, Washington, DC, USA, 3–12. DOI [10.1109/ICSTW.2017.6](https://doi.org/10.1109/ICSTW.2017.6) (Cited on page: 5)
- [17] Thomas Bach, Artur Andrzejak, Ralf Pannemans, and David Lo. 2017. The Impact of Coverage on Bug Density in a Large Industrial Software Project. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (Markham, Ontario, Canada) (ESEM 2017)*. IEEE Press, Washington, DC, USA, 307–313. ISBN 9781509040391 DOI [10.1109/ESEM.2017.44](https://doi.org/10.1109/ESEM.2017.44) (Cited on pages: 5 and 103)
- [18] Thomas Bach, Ralf Pannemans, and Artur Andrzejak. 2020. Determining Method-Call Sequences for Object Creation in C++. In *2020 IEEE International Conference on Software Testing, Verification and Validation (Porto, Portugal) (ICST 2020)*. IEEE Computer Society, Washington, DC, USA, 1–12. (Cited on page: 5)
- [19] Thomas Bach, Ralf Pannemans, Johannes Haeussler, and Artur Andrzejak. 2019. Dynamic Unit Test Extraction via Time-Travel Debugging for Test Cost Reduction. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings (Montreal, Quebec, Canada) (ICSE 2019)*. IEEE Press, Washington, DC, USA, 238–239. DOI [10.1109/ICSE-Companion.2019.00093](https://doi.org/10.1109/ICSE-Companion.2019.00093) (Cited on page: 5)
- [20] Thomas Bach, Ralf Pannemans, and Sascha Schwedes. 2018. Effects of an Economic Approach for Test Case Selection and Reduction for a Large Industrial Project. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE Computer Society, Washington, DC, USA, 374–379. DOI [10.1109/ICSTW.2018.00076](https://doi.org/10.1109/ICSTW.2018.00076) (Cited on pages: 3 and 5)
- [21] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (May 2015), 507–525. DOI [10.1109/TSE.2014.2372785](https://doi.org/10.1109/TSE.2014.2372785) (Cited on page: 8)
- [22] Earl T. Barr and Mark Marron. 2014. Tardis: Affordable Time-Travel Debugging in Managed Runtimes. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (Portland, Oregon, USA) (OOPSLA 2014)*. ACM, New York, NY, USA, 67–82. ISBN 9781450325851 DOI [10.1145/2660193.2660209](https://doi.org/10.1145/2660193.2660209) (Cited on page: 146)
- [23] Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. 2016. Time-Travel Debugging for JavaScript/Node.js. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (Seattle, WA, USA) (FSE 2016)*. ACM, New York, NY, USA, 1003–1007. ISBN 9781450342186 DOI [10.1145/2950290.2983933](https://doi.org/10.1145/2950290.2983933) (Cited on page: 146)
- [24] Adam B Barrett and Lionel Barnett. 2013. Granger Causality Is Designed to Measure Effect, Not Mechanism. *Frontiers in Neuroinformatics* 7 (2013), 6. DOI [10.3389/fninf.2013.00006](https://doi.org/10.3389/fninf.2013.00006) (Cited on page: 91)
- [25] Nauman bin Ali, Emelie Engström, Masoumeh Taromirad, Mohammad Reza Mousavi, Nasir Mehmood Minhas, Daniel Helgesson, Sebastian Kunze, and Mahsa Varshosaz. 2019. On the Search for Industry-Relevant Regression Testing Research. *Empirical Software Engineering* 24, 4 (Aug. 2019), 2020–2055. (Cited on page: 13)
- [26] Roderick Bloem, Robert Koenighofer, Franz Röck, and Michael Tautschnig. 2014. Automating Test-Suite Augmentation. In *Proceedings of the 2014 14th International Conference on Quality Software (QSIC 2014)*. IEEE Computer Society, USA, 67–72. ISBN 9781479971985 DOI [10.1109/QSIC.2014.40](https://doi.org/10.1109/QSIC.2014.40) (Cited on page: 103)
- [27] Vincent Blondeau, Anne Etien, Nicolas Anquetil, Sylvain Cresson, Pascal Croisy, and Stéphane Ducasse. 2017. Test Case Selection in Industry: An Analysis of Issues Related to Static Approaches. *Software Quality Journal* 25, 4 (Dec. 2017), 1203–1237. DOI [10.1007/s11219-016-9328-4](https://doi.org/10.1007/s11219-016-9328-4) (Cited on page: 118)
- [28] Barry William Boehm and Papaccio N. Papaccio. 1988. Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering* 14, 10 (Oct. 1988), 1462–1477. DOI [10.1109/32.6191](https://doi.org/10.1109/32.6191)

(Cited on pages: 61, 67, and 76)

- [29] Carl Boettiger. 2015. An Introduction to Docker for Reproducible Research. *ACM SIGOPS Operating Systems Review* 49, 1 (Jan. 2015), 71–79. DOI [10.1145/2723872.2723882](https://doi.org/10.1145/2723872.2723882) (Cited on page: 17)
- [30] Nicolas Bourgeois, Bruno Escoffier, and V Th Paschos. 2009. Efficient Approximation Of MIN SET COVER by Moderately Exponential Algorithms. *Theoretical Computer Science* 410, 21–23 (2009), 2184–2195. (Cited on page: 38)
- [31] George Edward Pelham Box, Gwilym Meirion Jenkins, Gregory Charles Reinsel, and Greta Marianne Ljung. 2015. *Time Series Analysis: Forecasting and Control* (fifth ed.). John Wiley and Sons, San Francisco, CA, USA. ISBN 0816211043 (Cited on pages: 72, 73, and 75)
- [32] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. *ACM SIGSOFT Software Engineering Notes* 27, 4 (July 2002), 123–133. DOI [10.1145/566171.566191](https://doi.org/10.1145/566171.566191) (Cited on page: 161)
- [33] Tim Bray, Jean Paoli, Christopher Michael Sperberg-McQueen, Eve Maler, and François Yergeau. 2008. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Technical Report. W3C. Retrieved 2020-01-10, archived by Internet Archive at <https://web.archive.org/web/20200110164217/https://www.w3.org/TR/2008/REC-xml-20081126/> from <http://www.w3.org/TR/2008/REC-xml-20081126/> (Cited on page: 158)
- [34] Derek L. Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. AAI0807735. (Cited on page: 29)
- [35] Yuriy Brun, Reid Holmes, Michael Dean Ernst, and David Notkin. 2011. Proactive Detection of Collaboration Conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (Szeged, Hungary) (ESEC/FSE 2011)*. ACM, New York, NY, USA, 168–178. ISBN 9781450304436 DOI [10.1145/2025113.2025139](https://doi.org/10.1145/2025113.2025139) (Cited on page: 20)
- [36] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (San Diego, California) (OSDI 2008)*. USENIX Association, Berkeley, CA, USA, 209–224. (Cited on page: 159)
- [37] Xia Cai and Michael R. Lyu. 2005. The Effect of Code Coverage on Fault Detection Under Different Testing Profiles. *SIGSOFT Software Engineering Notes* 30, 4 (May 2005), 1–7. DOI [10.1145/1082983.1083288](https://doi.org/10.1145/1082983.1083288) (Cited on page: 92)
- [38] Robert Callan, Farnaz Behrang, Alenka Zajic, Milos Prvulovic, and Alessandro Orso. 2016. Zero-Overhead Profiling via EM Emanations. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (Saarbrücken, Germany) (ISSTA 2016)*. ACM, New York, NY, USA, 401–412. ISBN 9781450343909 DOI [10.1145/2931037.2931065](https://doi.org/10.1145/2931037.2931065) (Cited on page: 30)
- [39] Gerardo Canfora, Michele Ceccarelli, Luigi Cerulo, and Massimiliano Di Penta. 2010. Using Multivariate Time Series and Association Rules to Detect Logical Change Coupling: An Empirical Study. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM 2010)*. IEEE Computer Society, USA, 1–10. ISBN 9781424486304 DOI [10.1109/ICSM.2010.5609732](https://doi.org/10.1109/ICSM.2010.5609732) (Cited on page: 93)
- [40] Jeffrey Carver, Letizia Jaccheri, Sandro Morasca, and Forrest Shull. 2003. Issues in Using Students in Empirical Studies in Software Engineering Education. In *Proceedings of the 9th International Symposium on Software Metrics (METRICS 2003)*. IEEE Computer Society, Washington, DC, USA, 239–249. ISBN 0769519873 (Cited on page: 179)
- [41] Stephen Cass. 2018. *The 2018 Top Programming Languages*. IEEE Spectrum. Retrieved 2020-01-10, archived by WebCite at <https://www.webcitation.org/76J2fuhpE> from <https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages> Rank 1 to 6: Python, C++, Java, C, C#, PHP. (Cited on pages: 160 and 182)
- [42] Pablo De Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. 2015. CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. *ACM Transactions on Architecture and Code Optimization* 12, 1 (April 2015), 1–24. DOI [10.1145/2724717](https://doi.org/10.1145/2724717) (Cited on pages: 141 and 159)
- [43] Sungdeok Cha, Richard N. Taylor, and Kyo Chul Kang. 2019. *Handbook of Software Engineering*. Springer Nature Switzerland, Gewerbestrasse 11, 6330 Cham, Switzerland. ISBN 9783030002619 DOI [10.1007/978-3-030-00262-6](https://doi.org/10.1007/978-3-030-00262-6) (Cited on page: 10)
- [44] Samy Chambi, Daniel Lemire, Robert Godin, Kamel Boukhalfa, Charles R. Allen, and Fangjin Yang. 2016. Optimizing Druid with Roaring Bitmaps. In *Proceedings of the 20th International Database Engineering (Montreal, QC, Canada) (IDEAS 2016)*. ACM, New York, NY, USA, 77–86. ISBN 9781450341189 DOI

- [10.1145/2938503.2938515](https://doi.org/10.1145/2938503.2938515) (Cited on page: 31)
- [45] John Joseph Chilenski. 2001. *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*. Technical Report. BOEING COMMERCIAL AIRPLANE CO SEATTLE WA. (Cited on pages: 28 and 97)
- [46] John Joseph Chilenski and Steven P. Miller. 1994. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal* 9, 5 (September 1994), 193–200. DOI [10.1049/sej.1994.0025](https://doi.org/10.1049/sej.1994.0025) (Cited on pages: 23, 28, and 97)
- [47] Eun-Hye Choi, Osamu Mizuno, and Yifan Hu. 2016. Code Coverage Analysis of Combinatorial Testing. In *Joint Proceedings of the 4th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2016) and 1st International Workshop on Technical Debt Analytics (TDA 2016) co-located with the 23rd Asia-Pacific Software Engineering Conference (APSEC 2016) (CEUR Workshop Proceedings)*. CEUR-WS.org, Hamilton, New Zealand, 43–49. (Cited on page: 104)
- [48] Vašek Chvátal. 1979. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research* 4, 3 (Aug. 1979), 233–235. DOI [10.1287/moor.4.3.233](https://doi.org/10.1287/moor.4.3.233) (Cited on page: 45)
- [49] Lori A. Clarke, Johnette Hassell, and Debra J. Richardson. 1982. A Close Look at Domain Testing. *IEEE Transactions on Software Engineering* SE-8, 4 (July 1982), 380–390. DOI [10.1109/TSE.1982.235572](https://doi.org/10.1109/TSE.1982.235572) (Cited on page: 10)
- [50] Reuven Cohen and Liran Katzir. 2008. The Generalized Maximum Coverage Problem. *Inform. Process. Lett.* 108, 1 (Sept. 2008), 15–22. DOI [10.1016/j.ipl.2008.03.017](https://doi.org/10.1016/j.ipl.2008.03.017) (Cited on page: 47)
- [51] Mike Cohn. 2009. *Succeeding with Agile: Software Development Using Scrum* (first ed.). Addison-Wesley Professional, USA. ISBN 0321579364, 9780321579362 (Cited on page: 3)
- [52] TIOBE Company. 2020. *TIOBE Programming Community Index*. TIOBE Company. Retrieved 2020-01-10, archived by Internet Archive at <https://web.archive.org/web/20200110152937/https://www.tiobe.com/tiobe-index/> from <https://www.tiobe.com/tiobe-index/> Rank 1 to 5: Java, C, Python, C++, C#. (Cited on pages: 160 and 182)
- [53] Thomas H. Cormen, Charles Eric Leiserson, Ronald Linn Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (third ed.). The MIT Press, Cambridge, Massachusetts London, England. ISBN 9780262033848 (Cited on pages: 53, 151, 168, and 169)
- [54] Cesar Couto. 2013. *Predicting Software Defects with Causality Tests*. Ph.D. Dissertation. Federal University of Minas Gerais. (Cited on pages: 80 and 92)
- [55] Cesar Couto, Pedro Pires, Marco Tulio Valente, Roberto S Bigonha, and Nicolas Anquetil. 2014. Predicting Software Defects With Causality Tests. *Journal of Systems and Software* 93 (2014), 24–41. (Cited on page: 80)
- [56] Cesar Couto, Christofer Silva, Marco Tulio Valente, Roberto Bigonha, and Nicolas Anquetil. 2012. Uncovering Causal Relationships between Software Metrics and Bugs. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering (CSMR 2012)*. IEEE Computer Society, USA, 223–232. ISBN 9780769546667 DOI [10.1109/CSMR.2012.31](https://doi.org/10.1109/CSMR.2012.31) (Cited on pages: 81 and 92)
- [57] Emilio Cruciani, Breno Miranda, Roberto Verdecchia, and Antonia Bertolino. 2019. Scalable Approaches for Test Suite Reduction. In *Proceedings of the 41st International Conference on Software Engineering (Montreal, Quebec, Canada) (ICSE 2019)*. IEEE Press, Washington, DC, USA, 419–429. DOI [10.1109/ICSE.2019.00055](https://doi.org/10.1109/ICSE.2019.00055) (Cited on page: 14)
- [58] Lajos Cseppentő and Zoltán Micskei. 2017. Evaluating Code-Based Test Input Generator Tools. *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on* 27 (2017), 1–24. Issue 6. DOI [10.1002/stvr.1627](https://doi.org/10.1002/stvr.1627) (Cited on page: 180)
- [59] Jacek Czerwinka. 2013. On Use of Coverage Metrics in Assessing Effectiveness of Combinatorial Test Designs. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2013)*. IEEE Computer Society, USA, 257–266. ISBN 9780769549934 DOI [10.1109/ICSTW.2013.76](https://doi.org/10.1109/ICSTW.2013.76) (Cited on page: 104)
- [60] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. 2019. A Snowballing Literature Study on Test Amplification. *Journal of Systems and Software* 157 (2019), 35. DOI [10.1016/j.jss.2019.110398](https://doi.org/10.1016/j.jss.2019.110398) (Cited on page: 103)
- [61] Al Danial. 2020. *cloc – Count Lines of Code*. Retrieved 2020-01-10, archived by WebCite at <http://www.webcitation.org/76J5fFUlo> from <https://github.com/AlDanial/cloc> (Cited on page: 172)
- [62] Martin Davis. 1983. *Computability and Unsolvability*. Dover Publications, Mineola, New York, USA. ISBN 9780486151069 (Cited on page: 167)
- [63] William Edwards Deming. 2000. *Out of the Crisis*. Massachusetts Institute of Technology, Center for Advanced Engineering Study, USA. ISBN 9780262541152 (Cited on page: 7)

- [64] AssertJ development team. 2020. *AssertJ - Fluent Assertions for Java*. Retrieved 2020-01-10, archived by Internet Archive at <https://web.archive.org/web/20200110152354/https://github.com/joel-costigliola/assertj-core> from <https://github.com/joel-costigliola/assertj-core> (Cited on page: 8)
- [65] Boost development team. 2018. *boost 1.66.0*. Retrieved 2020-01-10 from [https://dl.bintray.com/boostorg/release/1.66.0/source/boost\\_1\\_66\\_0.tar.gz](https://dl.bintray.com/boostorg/release/1.66.0/source/boost_1_66_0.tar.gz) (Cited on pages: 173 and 187)
- [66] CERN ROOT development team. 2018. *CERN ROOT 6.13/08*. CERN. Retrieved 2020-01-10 from [https://root.cern.ch/download/root\\_v6.13.08.source.tar.gz](https://root.cern.ch/download/root_v6.13.08.source.tar.gz) (Cited on pages: 173 and 187)
- [67] DynamoRIO development team. 2020. *drcov, a DynamoRIO Client Tool That Collects Code Coverage Information*. Retrieved 2020-01-10, archived by Internet Archive at [https://web.archive.org/web/20200110152248/https://dynamorio.org/docs/page\\_drcov.html](https://web.archive.org/web/20200110152248/https://dynamorio.org/docs/page_drcov.html) from [http://dynamorio.org/docs/page\\_drcov.html](http://dynamorio.org/docs/page_drcov.html) (Cited on pages: 21 and 29)
- [68] Firefox development team. 2018. *Firefox 55.0.3*. Mozilla. Retrieved 2020-01-10 from <https://archive.mozilla.org/pub/firefox/releases/55.0.3/source/firefox-55.0.3.source.tar.xz> (Cited on pages: 173 and 187)
- [69] LLVM Clang development team. 2018. *LLVM Clang 6.0.0*. Retrieved 2020-01-10 from <https://releases.llvm.org/6.0.0/llvm-6.0.0.src.tar.xz> (Cited on pages: 173 and 187)
- [70] MySQL development team. 2018. *MySQL 8.0.11*. Retrieved 2020-01-10 from <https://dev.mysql.com/get/Downloads/MySQL-8.0/mysql-boost-8.0.11.tar.gz> (Cited on pages: 173 and 187)
- [71] Pin development team. 2020. *Intel Pin - A Dynamic Binary Instrumentation Tool*. Retrieved 2020-01-10, archived by Internet Archive at <https://web.archive.org/web/20200110135215/https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool> from <https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool> (Cited on page: 150)
- [72] ScummVM development team. 2018. *ScummVM 2.0.0*. Retrieved 2020-01-10 from <https://www.scummvm.org/frs/scummvm/2.0.0/scummvm-2.0.0.tar.gz> (Cited on pages: 173 and 187)
- [73] Jeremy Dick, Elizabeth Hull, and Ken Jackson. 2017. *Requirements engineering* (fourth ed.). Springer International Publishing, Switzerland. ISBN 9783319610726 DOI 10.1007/978-3-319-61073-3 (Cited on page: 1)
- [74] William Dickinson, David Leon, and Andy Podgurski. 2001. Finding Failures by Cluster Analysis of Execution Profiles. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001* (Toronto, Ontario, Canada) (*ICSE 2001*). IEEE Computer Society, Washington, DC, USA, 339–348. ISBN 0769510507 DOI 10.1109/ICSE.2001.919107 (Cited on pages: 34 and 35)
- [75] Rheinhard Diestel. 2016. *Graph Theory*. Springer Nature, Heidelberger Platz 3, 14197 Berlin, Germany. ISBN 9783662536216 DOI 10.1007/978-3-662-53622-3 (Cited on page: 44)
- [76] Hyunsook Do. 2016. Recent Advances in Regression Testing Techniques. In *Advances in Computers*. Vol. 103. Elsevier, Amsterdam, Netherlands, 53–77. DOI 10.1016/bs.adcom.2016.04.004 (Cited on page: 13)
- [77] Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, and Gregg Rothermel. 2008. An Empirical Study of the Effect of Time Constraints on the Cost-Benefits of Regression Testing. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT 2008/FSE-16)*. ACM, New York, NY, USA, 71–82. ISBN 9781595939951 DOI 10.1145/1453101.1453113 (Cited on pages: 107 and 108)
- [78] Mark Dowson. 1997. The Ariane 5 Software Failure. *ACM SIGSOFT Software Engineering Notes* 22, 2 (March 1997), 84–85. DOI 10.1145/251880.251992 (Cited on page: 2)
- [79] Himanshu Shekhar Dutta. 2009. *Survey of Approximation Algorithms for Set Cover Problem*. Master’s thesis. University of North Texas. (Cited on page: 38)
- [80] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. 2006. Carving Differential Unit Test Cases from System Test Cases. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Portland, Oregon, USA) (*SIGSOFT 2006/FSE-14*). ACM, New York, NY, USA, 253–264. ISBN 1595934685 DOI 10.1145/1181775.1181806 (Cited on pages: 141 and 158)
- [81] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Matthew Jorde. 2009. Carving and Replaying Differential Unit Test Cases from System Test Cases. *IEEE Transactions on Software Engineering* 35, 1 (Jan. 2009), 29–45. DOI 10.1109/TSE.2008.103 (Cited on pages: 141, 143, and 158)
- [82] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2002. Test Case Prioritization: A Family of Empirical Studies. *IEEE Transactions on Software Engineering* 28, 2 (Feb. 2002), 159–182. DOI 10.1109/32.988497 (Cited on page: 107)

- [83] William R. Elmendorf. 1969. Controlling the Functional Testing of an Operating System. *IEEE Transactions on Systems Science and Cybernetics* 5, 4 (Oct 1969), 284–290. DOI [10.1109/TSSC.1969.300221](https://doi.org/10.1109/TSSC.1969.300221) (Cited on page: 23)
- [84] Jakob Engblom. 2012. A Review of Reverse Debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*. IEEE, IEEE Computer Society, Washington, DC, USA, 1–6. (Cited on pages: 143 and 146)
- [85] Michael G. Epitropakis, Shin Yoo, Mark Harman, and Edmund K. Burke. 2015. Empirical Evaluation of Pareto Efficient Multi-Objective Regression Test Case Prioritisation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 234–245. ISBN 9781450336208 DOI [10.1145/2771783.2771788](https://doi.org/10.1145/2771783.2771788) (Cited on pages: 51, 108, and 126)
- [86] Jens Grabowski Fabian Trautsch. 2017. Are There Any Unit Tests? An Empirical Study on Unit Testing in Open Source Python Projects. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST 2017)*. IEEE Computer Society, Washington, DC, USA, 207–218. DOI [10.1109/ICST.2017.26](https://doi.org/10.1109/ICST.2017.26) (Cited on page: 16)
- [87] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (Vasteras, Sweden) (ASE 2014)*. Association for Computing Machinery, New York, NY, USA, 313–324. ISBN 9781450330138 DOI [10.1145/2642937.2642982](https://doi.org/10.1145/2642937.2642982) (Cited on pages: 60, 77, 78, and 90)
- [88] Sheikh Umar Farooq, S.M.K. Quadri, and Nesar Ahmad. 2017. A Replicated Empirical Study to Evaluate Software Testing Methods. *Journal of Software: Evolution and Process* 29, 9 (2017), 1–22. DOI [10.1002/smr.1883](https://doi.org/10.1002/smr.1883) (Cited on page: 10)
- [89] Michael Felderer and Ina Schieferdecker. 2014. A Taxonomy of Risk-Based Testing. *International Journal on Software Tools for Technology Transfer* 16, 5 (jul 2014), 559–568. DOI [10.1007/s10009-014-0332-3](https://doi.org/10.1007/s10009-014-0332-3) (Cited on pages: 54 and 118)
- [90] Robert Feldt and Felix Dobsław. 2019. Towards Automated Boundary Value Testing with Program Derivatives and Search. In *Search-Based Software Engineering*. Springer International Publishing, Cham, 155–163. ISBN 9783030274559 DOI [10.1007/978-3-030-27455-9\\\_11](https://doi.org/10.1007/978-3-030-27455-9\_11) (Cited on page: 10)
- [91] Robert Feldt and Ana Magazinius. 2010. Validity Threats in Empirical Software Engineering Research – An Initial Survey. In *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010)*. SEKE, Redwood City, San Francisco Bay, CA, USA, 374–379. (Cited on page: 57)
- [92] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (Szeged, Hungary) (ESEC/FSE 2011)*. ACM, New York, NY, USA, 416–419. ISBN 9781450304436 DOI [10.1145/2025113.2025179](https://doi.org/10.1145/2025113.2025179) (Cited on pages: 161, 162, 180, and 182)
- [93] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Record* 40, 4 (Jan. 2012), 45–51. DOI [10.1145/2094114.2094126](https://doi.org/10.1145/2094114.2094126) (Cited on page: 15)
- [94] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *Bulletin of the Technical Committee on Data Engineering / IEEE Computer Society* 35, 1 (2012), 28–33. (Cited on page: 15)
- [95] Winrich A. Freiwald, Pedro Valdes, Jorge Bosch, Rolando Biscay, Juan Carlos Jimenez, Luis Manuel Rodriguez, Valia Rodriguez, Andreas K Kreiter, and Wolf Singer. 1999. Testing Non-Linearity and Directedness of Interactions Between Neural Groups in the Macaque Inferotemporal Cortex. *Journal of neuroscience methods* 94, 1 (1999), 105–119. (Cited on page: 74)
- [96] Wayne Arthur Fuller. 1996. *Introduction to Statistical Time Series*. Wiley, USA. (Cited on pages: 72 and 81)
- [97] Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. William H. Freeman and Company, USA. ISBN 0716710455 (Cited on page: 51)
- [98] Pranav Garg, Franjo Ivancic, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. 2013. Feedback-Directed Unit Test Generation for C/C++ Using Concolic Execution. In *Proceedings of the 2013 International Conference on Software Engineering (San Francisco, CA, USA) (ICSE 2013)*. IEEE Press, Piscataway, NJ, USA, 132–141. ISBN 9781467330763 (Cited on pages: 159 and 181)
- [99] John Geweke. 1984. Inference and Causality in Economic Time Series Models. In *Handbook of Econometrics*. Handbook of Econometrics, Vol. 2. Elsevier, Amsterdam, Netherlands, 1101–1144. (Cited on page: 90)



- [100] GitHub. 2018. *The Fifteen Most Popular Languages on GitHub by Opened Pull Request*. GitHub Inc. Retrieved 2020-01-10, archived by WebCite at <https://www.webcitation.org/76J2L0Td3> from <https://octoverse.github.com/2018/> Rank 1 to 6: Javascript, Python, Java, Ruby, PHP, C++. (Cited on pages: 160 and 182)
- [101] Patrice Godefroid. 2014. Micro Execution. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) (*ICSE 2014*). ACM, New York, NY, USA, 539–549. ISBN 9781450327565 DOI [10.1145/2568225.2568273](https://doi.org/10.1145/2568225.2568273) (Cited on page: 29)
- [102] David Goldberg. 1991. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys (CSUR)* 23, 1 (1991), 5–48. (Cited on page: 8)
- [103] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code Coverage for Suite Evaluation by Developers. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 72–82. ISBN 9781450327565 DOI [10.1145/2568225.2568278](https://doi.org/10.1145/2568225.2568278) (Cited on pages: 13, 70, and 92)
- [104] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Mutations: How Close Are They to Real Faults? In *Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering (ISSRE 2014)*. IEEE Computer Society, Washington, DC, USA, 189–200. ISBN 9781479960330 DOI [10.1109/ISSRE.2014.40](https://doi.org/10.1109/ISSRE.2014.40) (Cited on pages: 70 and 92)
- [105] Clive WJ Granger and Paul Newbold. 1974. Spurious Regressions in Econometrics. *Journal of econometrics* 2, 2 (1974), 111–120. (Cited on pages: 72 and 73)
- [106] Clive William John Granger. 1969. Investigating Causal Relations by Econometric Models and Cross-Spectral Methods. *Econometrica* 37, 3 (1969), 424–438. (Cited on pages: 69, 71, 72, and 73)
- [107] Alfred Gray, Elsa Abbena, and Simon Salamon. 2006. *Modern Differential Geometry of Curves and Surfaces with Mathematica* (third ed.). Chapman and Hall/CRC, USA. ISBN 1584884487 (Cited on page: 34)
- [108] Lucas Gren. 2018. Standards of Validity and the Validity of Standards in Behavioral Software Engineering Research: The Perspective of Psychological Test Theory. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (Oulu, Finland) (ESEM 2018)*. ACM, New York, NY, USA, 1–4. ISBN 9781450358231 DOI [10.1145/3239235.3267437](https://doi.org/10.1145/3239235.3267437) (Cited on page: 57)
- [109] Mats Grindal, Jeff Offutt, and Sten F Andler. 2005. Combination Testing Strategies: A Survey. *Software Testing, Verification and Reliability* 15, 3 (2005), 167–199. (Cited on page: 10)
- [110] Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. 2014. Coverage and Its Discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, Oregon, USA) (*Onward! 2014*). ACM, Association for Computing Machinery, New York, NY, USA, 255–268. ISBN 9781450332101 DOI [10.1145/2661136.2661157](https://doi.org/10.1145/2661136.2661157) (Cited on page: 56)
- [111] Alex David Groce. 2005. *Error Explanation and Fault Localization with Distance Metrics*. Ph.D. Dissertation. Carnegie Mellon University. ISBN 0542015544 (Cited on page: 34)
- [112] Mário Luís Guimarães and António Rito Silva. 2012. Improving Early Detection of Software Merge Conflicts. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) (*ICSE 2012*). IEEE Press, IEEE Press, Washington, DC, USA, 342–352. ISBN 9781467310673 DOI [10.1109/ICSE.2012.6227180](https://doi.org/10.1109/ICSE.2012.6227180) (Cited on page: 20)
- [113] James Douglas Hamilton. 1994. *Time Series Analysis*. Princeton University Press, USA. (Cited on page: 73)
- [114] Mark Harman. 2010. Why Source Code Analysis and Manipulation Will Always be Important. In *Proceedings of the 10th International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, Washington, DC, USA, 7–19. ISBN 9780769541785 DOI [10.1109/SCAM.2010.28](https://doi.org/10.1109/SCAM.2010.28) (Cited on page: 24)
- [115] Kelly J. Hayhurst and Dan S. Veerhusen. 2001. A Practical Approach to Modified Condition/Decision Coverage. In *20th DASC. 20th Digital Avionics Systems Conference*, Vol. 1. IEEE Computer Society, Washington, DC, USA, 1–10. DOI [10.1109/DASC.2001.963305](https://doi.org/10.1109/DASC.2001.963305) (Cited on page: 103)
- [116] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The Art of Testing Less Without Sacrificing Quality. In *Proceedings of the 37th International Conference on Software Engineering* (Florence, Italy) (*ICSE 2015*). IEEE Press, Piscataway, NJ, USA, 483–493. ISBN 9781479919345 (Cited on pages: 3 and 12)
- [117] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It’s Not a Bug, It’s a Feature: How Misclassification Impacts Bug Prediction. In *Proceedings of the 2013 international conference on software engineering* (San

- Francisco, CA, USA) (*ICSE 2013*). IEEE Press, IEEE Computer Society, Washington, DC, USA, 392–401. ISBN 9781467330763 (Cited on page: 89)
- [118] Kim Herzig, Sascha Just, and Andreas Zeller. 2016. The Impact of Tangled Code Changes on Defect Prediction Models. *Empirical Software Engineering* 21, 2 (2016), 303–336. (Cited on page: 89)
- [119] Kim Herzig and Andreas Zeller. 2013. The Impact of Tangled Code Changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (San Francisco, CA, USA) (*MSR 2013*). IEEE Press, Piscataway, NJ, USA, 121–130. ISBN 9781467329361 (Cited on page: 67)
- [120] Michael Hilton, Jonathan Bell, and Darko Marinov. 2018. A Large-Scale Study of Test Coverage Evolution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) (*ASE 2018*). ACM, New York, NY, USA, 53–63. ISBN 9781450359375 DOI [10.1145/3238147.3238183](https://doi.org/10.1145/3238147.3238183) (Cited on page: 23)
- [121] Richard Hipp et al. 2020. *SQLite*. Retrieved 2020-01-10, archived by Internet Archive at <https://web.archive.org/web/20200110153303/https://www.sqlite.org/index.html> from <https://www.sqlite.org/index.html> Version 3.24.0 from 2018-06-04. (Cited on page: 144)
- [122] Joy W Hollén and Patrick S Zacarias. 2015. *Exploring Code Coverage in Software Testing and its Correlation with Software Quality; A Systematic Literature Review*. Master’s thesis. University of Gothenburg. (Cited on page: 23)
- [123] Christian Hovy and Julian Kunkel. 2016. Towards Automatic and Flexible Unit Test Generation for Legacy HPC Code. In *Proceedings of the Fourth International Workshop on Software Engineering for HPC in Computational Science and Engineering* (Salt Lake City, Utah) (*SE-HPCCSE 2016*). IEEE Press, Piscataway, NJ, USA, 42–49. ISBN 9781509052240 DOI [10.1109/SE-HPCCSE.2016.6](https://doi.org/10.1109/SE-HPCCSE.2016.6) (Cited on pages: 141, 143, and 159)
- [124] William E. Howden. 1976. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering* SE-2, 3 (September 1976), 208–215. DOI [10.1109/TSE.1976.233816](https://doi.org/10.1109/TSE.1976.233816) (Cited on page: 2)
- [125] William E. Howden. 1978. Theoretical and Empirical Studies of Program Testing. *IEEE Transactions on Software Engineering* SE-4, 4 (July 1978), 293–298. DOI [10.1109/TSE.1978.231514](https://doi.org/10.1109/TSE.1978.231514) (Cited on page: 8)
- [126] Sanqing Hu, Yu Cao, Jianhai Zhang, Wanzeng Kong, Kun Yang, Yanbin Zhang, and Xun Li. 2012. More Discussions for Granger Causality and New Causality Measures. *Cognitive neurodynamics* 6, 1 (2012), 33–42. (Cited on page: 91)
- [127] Sanqing Hu, Guojun Dai, Gregory A Worrell, Qionghai Dai, and Hualou Liang. 2011. Causality Analysis of Neural Connectivity: Critical Examination of Existing Methods and Advances of New Methods. *IEEE transactions on neural networks* 22, 6 (2011), 829–844. (Cited on page: 74)
- [128] Xueqing Hu, Sanqing Hu, Jianhai Zhang, Wanzeng Kong, and Yu Cao. 2016. A Fatal Drawback of the Widely Used Granger Causality in Neuroscience. In *2016 Sixth International Conference on Information Science and Technology (ICIST)*. IEEE, IEEE Computer Society, Washington, DC, USA, 61–65. DOI [10.1109/ICIST.2016.7483386](https://doi.org/10.1109/ICIST.2016.7483386) (Cited on page: 91)
- [129] Thomas Huckle and Tobias Neckel. 2019. *Bits and Bugs: A Scientific and Historical Review of Software Failures in Computational Science*. Society for Industrial and Applied Mathematics, Philadelphia, PA. ISBN 9781611975550 DOI [10.1137/1.9781611975567](https://doi.org/10.1137/1.9781611975567) (Cited on page: 2)
- [130] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) (*ICSE 2014*). ACM, New York, NY, USA, 435–445. ISBN 9781450327565 DOI [10.1145/2568225.2568271](https://doi.org/10.1145/2568225.2568271) (Cited on pages: 56, 58, 70, 91, 92, and 103)
- [131] Intel Corporation. 2019. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Technical Report. Intel Corporation. Intel number: 248966-042b. (Cited on pages: 23 and 30)
- [132] Intel Corporation. 2019. *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Technical Report. Intel Corporation. Intel number: 325462-071US. (Cited on pages: 23, 25, and 27)
- [133] ISO. 2011. *Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models*. Technical Report ISO/IEC 25010:2011. International Organization for Standardization, Geneva, Switzerland. (Cited on pages: 1 and 9)
- [134] ISO. 2011. *Systems and Software Engineering – Vocabulary*. Technical Report ISO/IEC/IEEE 24765:2017. International Organization for Standardization, Geneva, Switzerland. (Cited on page: 16)
- [135] ISO. 2015. *Information Technology – Vocabulary*. Technical Report ISO/IEC 2382:2015. International Organization for Standardization, Geneva, Switzerland. (Cited on page: 1)
- [136] ISO. 2015. *Quality Management Systems – Fundamentals and Vocabulary*. Technical Report ISO 9000:2015. International Organization for Standardization, Geneva, Switzerland. (Cited on pages: 1, 7, and 8)

- [137] ISO. 2017. *Information Technology – Universal Coded Character Set (UCS)*. Technical Report ISO/IEC 10646:2017. International Organization for Standardization, Geneva, Switzerland. (Cited on page: 24)
- [138] ISO. 2017. *Programming Languages – C++*. Technical Report ISO/IEC 14882:2017. International Organization for Standardization, Geneva, Switzerland. (Cited on pages: 146, 157, 161, 162, 165, 166, 167, 170, 171, 172, 173, 177, and 184)
- [139] ISTQB. 2020. *International Software Testing Qualifications Board (ISTQB)*. ISTQB. Retrieved 2020-01-10, archived by Internet Archive at <https://web.archive.org/web/20200110161249/https://www.istqb.org/> from <http://www.istqb.org/> (Cited on page: 61)
- [140] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code Coverage at Google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Tallinn, Estonia) (ESEC/FSE 2019)*. ACM, New York, NY, USA, 955–963. ISBN 9781450355728 DOI 10.1145/3338906.3340459 (Cited on page: 23)
- [141] Paul Jaccard. 1902. Lois de Distribution Florale dans la Zone Alpine. *Bull Soc Vaudoise Sci Nat* 38 (1902), 69–130. (Cited on page: 127)
- [142] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. 2010. OCAT: Object Capture-Based Automated Testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (Trento, Italy) (ISSTA 2010)*. ACM, New York, NY, USA, 159–170. ISBN 9781605588230 DOI 10.1145/1831708.1831729 (Cited on pages: 141, 159, 161, 180, and 182)
- [143] Bo Jiang, Zhenyu Zhang, Tsun-Him Tse, and Tsong Yueh Chen. 2009. How Well Do Test Case Prioritization Techniques Support Statistical Fault Localization. In *Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference – Volume 01 (COMPSAC 2009)*. IEEE Computer Society, Washington, DC, USA, 99–106. ISBN 9780769537269 DOI 10.1109/COMPSAC.2009.23 (Cited on pages: 34 and 35)
- [144] Capers Jones. 2007. *Estimating Software Costs: BRinging Realism to Estimating*. McGraw-Hill Companies New York, New York, USA. ISBN 9780071483001 (Cited on page: 12)
- [145] James A. Jones and Mary Jean Harrold. 2003. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. *IEEE Transactions on Software Engineering* 29, 3 (March 2003), 195–209. DOI 10.1109/TSE.2003.1183927 (Cited on pages: 28 and 97)
- [146] Shrinivas Joshi and Alessandro Orso. 2007. SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions. In *2007 IEEE International Conference on Software Maintenance*. IEEE Computer Society, Washington, DC, USA, 234–243. DOI 10.1109/ICSM.2007.4362636 (Cited on pages: 141 and 158)
- [147] René Just, Darioush Jalali, Laura Inozemtseva, Michael Dean Ernst, Reid Holmes, and Gordon Fraser. 2014. 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 (Hong Kong, China) (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 654–665. ISBN 9781450330565 DOI 10.1145/2635868.2635929 (Cited on page: 92)
- [148] Richard Manning Karp. 1972. Reducibility Among Combinatorial Problems. In *Proceedings of a Symposium on the Complexity of Computer Computations* (IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA). Springer US, Boston, MA, 85–103. ISBN 9781468420012 DOI 10.1007/978-1-4684-2001-2\_9 (Cited on pages: 37, 47, and 98)
- [149] Rafaqut Kazmi, Dayang N. A. Jawawi, Radziah Mohamad, and Imran Ghani. 2017. Effective Regression Test Case Selection: A Systematic Literature Review. *Comput. Surveys* 50, 2 (May 2017), 32. DOI 10.1145/3057269 (Cited on pages: 13 and 159)
- [150] Rafaqut Kazmi, Dayang N. A. Jawawi, Radziah Mohamad, and Imran Ghani. 2017. Effective Regression Test Case Selection: A Systematic Literature Review. *Comput. Surveys* 50, 2 (May 2017), 1–32. DOI 10.1145/3057269 (Cited on page: 118)
- [151] Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. *Knapsack Problems*. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 9783642073113 9783540247777 (Cited on page: 120)
- [152] Brian Wilson Kernighan and Phillip James Plauger. 1978. *The Elements of Programming Style* (second ed.). McGraw-Hill, New York, USA. ISBN 9780070342071 (Cited on pages: 24 and 140)
- [153] Saif Ur Rehman Khan, Sai Peck Lee, Nadeem Javaid, and Wadood Abdul. 2018. A systematic review on test suite reduction: Approaches, experiment’s quality evaluation, and guidelines. *IEEE Access* 6 (2018), 11816–11841. (Cited on page: 13)
- [154] Muhammad Khatibsyarhini, Mohd Adham Isa, Dayang NA Jawawi, and Rooster Tumeng. 2018. Test Case Prioritization Approaches in Regression Testing: A Systematic Literature Review. *Information and Software Technology* 93 (2018), 74–93. (Cited on page: 13)

- [155] Yit Phang Khoo, Jeffrey S Foster, and Michael Hicks. 2013. Expositor: Scriptable Time-Travel Debugging With First-Class Traces. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) (*ICSE 2013*). IEEE Press, IEEE Computer Society, Washington, DC, USA, 352–361. ISBN 9781467330763 DOI [10.1109/ICSE.2013.6606581](https://doi.org/10.1109/ICSE.2013.6606581) (Cited on page: 146)
- [156] Samir Khuller, Anna Moss, and Joseph Seffi Naor. 1999. The Budgeted Maximum Coverage Problem. *Inform. Process. Lett.* 70, 1 (April 1999), 39–45. DOI [10.1016/S0020-0190\(99\)00031-9](https://doi.org/10.1016/S0020-0190(99)00031-9) (Cited on pages: 48 and 49)
- [157] Youngsung Kim, John Dennis, Christopher Kerr, Raghu Raj Prasanna Kumar, Amogh Simha, Allison Baker, and Sheri Mickelson. 2016. KGEN: A Python Tool for Automated FORTRAN Kernel Generation and Verification. *Procedia Computer Science* 80 (2016), 1450–1460. (Cited on pages: 141 and 159)
- [158] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 2015. Code Coverage and Test Suite Effectiveness: Empirical Study With Real Bugs in Large Systems. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering* (Montreal, QC, Canada) (*SANER 2015*). IEEE Computer Society, Washington, DC, USA, 560–564. DOI [10.1109/SANER.2015.7081877](https://doi.org/10.1109/SANER.2015.7081877) (Cited on pages: 56 and 91)
- [159] Pavneet Singh Kochhar, Ferdian Thung, David Lo, and Julia Lawall. 2014. An Empirical Study on the Adequacy of Testing in Open Source Projects. In *Proceedings of the 2014 21st Asia-Pacific Software Engineering Conference (APSEC 2014)*. IEEE Computer Society, USA, 215–222. ISBN 9781479974269 DOI [10.1109/APSEC.2014.42](https://doi.org/10.1109/APSEC.2014.42) (Cited on page: 92)
- [160] Gary Koop. 2006. *Analysis of Financial Data*. Wiley, USA. (Cited on page: 73)
- [161] David Kordalewski. 2013. *New Greedy Heuristics For Set Cover and Set Packing*. Master’s thesis. University of Toronto. (Cited on pages: 38 and 44)
- [162] Bernhard Korte and Jens Vygen. 2018. *Combinatorial Optimization: Theory and Algorithms* (sixth ed.). Springer Publishing Company, Incorporated, Heidelberg, Germany. ISBN 3662560380, 9783662560389 DOI [10.1007/3-540-29297-7](https://doi.org/10.1007/3-540-29297-7) (Cited on pages: 37, 50, and 51)
- [163] Jon A. Krosnick and Stanley Presser. 2009. Question and Questionnaire Design. In *Handbook of Survey Research* (second ed.), Peter V. Marsden and James D. Wright (Eds.). Emerald Group Publishing, Bingley, UK, Chapter 9, 439–455. (Cited on page: 162)
- [164] David Richard Kuhn, Raghu N. Kacker, and Yu Lei. 2010. Practical Combinatorial Testing. *NIST special Publication* 800, 142 (2010), 142. (Cited on page: 11)
- [165] David Richard Kuhn, Yu Lei, and Raghu N. Kacker. 2008. Practical Combinatorial Testing: Beyond Pairwise. *It Professional* 10, 3 (2008), 19–23. (Cited on page: 11)
- [166] David Richard Kuhn, Dolores R Wallace, and Albert M. Gallo. 2004. Software Fault Interactions and Implications for Software Testing. *IEEE transactions on software engineering* 30, 6 (2004), 418–421. (Cited on page: 11)
- [167] Filip Kriikava and Jan Vitek. 2018. Tests from Traces: Automated Unit Test Extraction for R. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) (*ISSTA 2018*). ACM, New York, NY, USA, 232–241. ISBN 9781450356992 DOI [10.1145/3213846.3213863](https://doi.org/10.1145/3213846.3213863) (Cited on pages: 141, 143, and 158)
- [168] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the Cost of Regression Testing in Practice: A Study of Java Projects Using Continuous Integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) (*ESEC/FSE 2017*). ACM, New York, NY, USA, 821–830. ISBN 9781450351058 DOI [10.1145/3106237.3106288](https://doi.org/10.1145/3106237.3106288) (Cited on page: 139)
- [169] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) (*ISSTA 2019*). Association for Computing Machinery, New York, NY, USA, 101–111. ISBN 9781450362245 DOI [10.1145/3293882.3330570](https://doi.org/10.1145/3293882.3330570) (Cited on page: 139)
- [170] Michele Lanza. 2001. The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques. In *Proceedings of the 4th International Workshop on Principles of Software Evolution* (Vienna, Austria) (*IWPSE 2001*). Association for Computing Machinery, New York, NY, USA, 37–42. ISBN 1581135084 DOI [10.1145/602461.602467](https://doi.org/10.1145/602461.602467) (Cited on page: 80)
- [171] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Washington, DC, USA, 75–86. DOI [10.1109/CGO.2004.1281665](https://doi.org/10.1109/CGO.2004.1281665) (Cited on pages: 28, 147, 159, and 166)
- [172] Yoon-Ju Lee and Mary Hall. 2005. A Code Isolator: Isolating Code Fragments from Large Programs.

- In *Proceedings of the 17th International Conference on Languages and Compilers for High Performance Computing* (West Lafayette, IN) (*LCPC 2004*). Springer-Verlag, Berlin, Heidelberg, 164–178. ISBN 9783540280095 DOI [10.1007/11532378\\_13](https://doi.org/10.1007/11532378_13) (Cited on page: 159)
- [173] Meir Manny Lehman. 1980. Programs, Life Cycles, and Laws of Software Evolution. *Proc. IEEE* 68, 9 (1980), 1060–1076. (Cited on page: 72)
- [174] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O’Hara, François Saint-Jacques, and Gregory Ssi-Yan-Kai. 2018. Roaring Bitmaps: Implementation of an Optimized Software Library. *Software: Practice and Experience* 48, 4 (Jan 2018), 867–895. DOI [10.1002/spe.2560](https://doi.org/10.1002/spe.2560) (Cited on page: 31)
- [175] David Leon and Andy Podgurski. 2003. A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE 2003)*. IEEE, IEEE Computer Society, USA, 442–453. ISBN 0769520073 DOI [10.1109/ISSRE.2003.1251065](https://doi.org/10.1109/ISSRE.2003.1251065) (Cited on pages: 34 and 35)
- [176] Maurizio Leotta, Maura Cerioli, Dario Olinas, and Filippo Ricca. 2019. Hamcrest vs AssertJ: An Empirical Assessment of Tester Productivity. In *International Conference on the Quality of Information and Communications Technology*. Springer, Springer International Publishing, Cham, 161–176. ISBN 9783030292386 DOI [10.1007/978-3-030-29238-6\\_12](https://doi.org/10.1007/978-3-030-29238-6_12) (Cited on page: 8)
- [177] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. 2011. KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs. In *Computer Aided Verification*, Shaz Gopalakrishnan, Ganeshand Qadeer (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 609–615. ISBN 9783642221101 (Cited on pages: 159 and 181)
- [178] Zheng Li, Mark Harman, and Robert M. Hierons. 2007. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering* 33, 4 (April 2007), 225–237. DOI [10.1109/TSE.2007.38](https://doi.org/10.1109/TSE.2007.38) (Cited on page: 107)
- [179] Helmut Lütkepohl. 2005. *New Introduction to Multiple Time Series Analysis*. Springer-Verlag Berlin Heidelberg, Germany. ISBN 9783540401728 DOI [10.1007/978-3-540-27752-1](https://doi.org/10.1007/978-3-540-27752-1) (Cited on pages: 72, 73, and 75)
- [180] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) (*PLDI 2005*). ACM, New York, NY, USA, 190–200. ISBN 1595930566 DOI [10.1145/1065010.1065034](https://doi.org/10.1145/1065010.1065034) (Cited on page: 150)
- [181] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) (*FSE 2014*). ACM, New York, NY, USA, 643–653. ISBN 9781450330565 DOI [10.1145/2635868.2635920](https://doi.org/10.1145/2635868.2635920) (Cited on pages: 90, 108, 139, and 142)
- [182] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramlér. 2015. GRT: Program-Analysis-Guided Random Testing. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE 2015)*. IEEE Computer Society, Washington, DC, USA, 212–223. ISBN 9781509000258 DOI [10.1109/ASE.2015.49](https://doi.org/10.1109/ASE.2015.49) (Cited on pages: 161, 180, and 182)
- [183] Lei Ma, Cheng Zhang, Bing Yu, and Hiroyuki Sato. 2017. An Empirical Study on the Effects of Code Visibility on Program Testability. *Software Quality Journal* 25, 3 (Sept. 2017), 951–978. DOI [10.1007/s11219-016-9340-8](https://doi.org/10.1007/s11219-016-9340-8) (Cited on page: 167)
- [184] Mateusz Machalica, Alex Samykin, Meredith Porth, and Satish Chandra. 2019. Predictive Test Selection. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice* (Montreal, Quebec, Canada) (*ICSE-SEIP 2019*). IEEE Press, Piscataway, NJ, USA, 91–100. DOI [10.1109/ICSE-SEIP.2019.00018](https://doi.org/10.1109/ICSE-SEIP.2019.00018) (Cited on pages: 3 and 14)
- [185] Henry B Mann and Donald R Whitney. 1947. On a Test of Whether One of Two Random Variables Is Stochastically Larger Than the Other. *Annals of Mathematical Statistics* 18, 1 (1947), 50–60. (Cited on page: 86)
- [186] Paul Marinescu, Petr Hosek, and Cristian Cadar. 2014. Covrig: A Framework for the Analysis of Code, Test, and Coverage Evolution in Real Software. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 93–104. ISBN 9781450326452 DOI [10.1145/2610384.2610419](https://doi.org/10.1145/2610384.2610419) (Cited on pages: 136 and 137)
- [187] Robert Cecil Martin. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA. ISBN 0132350882, 9780132350884 (Cited on pages: 8 and 68)
- [188] Norman May, Alexander Böhm, and Wolfgang Lehner. 2017. SAP HANA – The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads. In *Datenbanksysteme für Business, Tech-*

- nologie und Web (BTW 2017)*. Gesellschaft für Informatik, Bonn, Germany, 545–563. ISBN 9783885796596 (Cited on page: 15)
- [189] Mariusz Maziarz. 2015. A Review of the Granger-Causality Fallacy. *The journal of philosophical economics: Reflections on economic and social issues* 8, 2 (2015), 86–105. (Cited on pages: 74 and 75)
- [190] William M McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107. (Cited on page: 12)
- [191] Atif Memon, Bao Nguyen, Eric Nickell, John Micco, Sanjeev Dhanda, Rob Siemborski, and Zebao Gao. 2017. Taming Google-Scale Continuous Testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track* (Buenos Aires, Argentina) (*ICSE-SEIP 2017*). IEEE Press, Washington, DC, USA, 233–242. ISBN 9781538627174 DOI [10.1109/ICSE-SEIP.2017.16](https://doi.org/10.1109/ICSE-SEIP.2017.16) (Cited on pages: 3, 14, 93, and 118)
- [192] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux journal* 2014, 239 (2014), 2. (Cited on page: 17)
- [193] John Micco. 2016. *Flaky Tests at Google and How We Mitigate Them*. Google. Retrieved 2020-01-10, archived by WebCite at <http://www.webcitation.org/78RWFvFQ> from <https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html> (Cited on page: 139)
- [194] Audris Mockus, Nachiappan Nagappan, and Trung Dinh-Trong. 2009. Test Coverage and Post-Verification Defects: A Multiple Case Study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM 2009)*. IEEE Computer Society, USA, 291–301. ISBN 9781424448425 DOI [10.1109/ESEM.2009.5315981](https://doi.org/10.1109/ESEM.2009.5315981) (Cited on pages: 55, 68, 70, and 91)
- [195] Salim Ali Khan Mohammad, Sathvik Vamshi Valepe, Subhrakanta Panda, and B.S.A.S. Rajita. 2019. A Comparative Study of the Effectiveness of Meta-Heuristic Techniques in Pairwise Testing. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, IEEE Computer Society, Washington, DC, USA, 91–96. DOI [10.1109/COMPSAC.2019.00022](https://doi.org/10.1109/COMPSAC.2019.00022) (Cited on page: 11)
- [196] Jefferson Seide Molléri, Kai Petersen, and Emilia Mendes. 2019. CERSE – Catalog for empirical research in software engineering: A Systematic mapping study. *Information & Software Technology* 105 (2019), 117–149. DOI [10.1016/j.infsof.2018.08.008](https://doi.org/10.1016/j.infsof.2018.08.008) (Cited on page: 57)
- [197] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. 2018. *Handbook of Floating-Point Arithmetic* (second ed.). Birkhäuser, Basel, Switzerland. 1–627 pages. ISBN 9783319765259 DOI [10.1007/978-3-319-76526-6](https://doi.org/10.1007/978-3-319-76526-6) (Cited on page: 8)
- [198] Nachiappan Nagappan and Thomas Ball. 2005. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA) (*ICSE 2005*). Association for Computing Machinery, New York, NY, USA, 284–292. ISBN 1581139632 DOI [10.1145/1062455.1062514](https://doi.org/10.1145/1062455.1062514) (Cited on page: 91)
- [199] Akbar Siami Namin and James H. Andrews. 2009. The Influence of Size and Coverage on Test Suite Effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (Chicago, IL, USA) (*ISSTA 2009*). ACM, New York, NY, USA, 57–68. ISBN 9781605583389 (Cited on page: 56)
- [200] Akbar Siami Namin and Sahitya Kakarla. 2011. The Use of Mutation in Testing Experiments and Its Sensitivity to External Threats. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) (*ISSTA 2011*). Association for Computing Machinery, New York, NY, USA, 342–352. ISBN 9781450305624 DOI [10.1145/2001420.2001461](https://doi.org/10.1145/2001420.2001461) (Cited on pages: 70 and 92)
- [201] Mark Nelson and Jean-Loup Gailly. 1995. *The Data Compression Book* (second ed.). MIS:Press, USA. ISBN 1558514341 (Cited on page: 123)
- [202] Iulia Nica, Gerhard Jakob, Kathrin Juhart, and Franz Wotawa. 2017. Results of a Comparative Study of Code Coverage Tools in Computer Vision. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE Computer Society, Washington, DC, USA, 36–37. DOI [10.1109/ICSTW.2017.10](https://doi.org/10.1109/ICSTW.2017.10) (Cited on page: 23)
- [203] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *ACM Computing Surveys (CSUR)* 43, 2 (2011), 11. DOI [10.1145/1883612.1883618](https://doi.org/10.1145/1883612.1883618) (Cited on page: 11)
- [204] James Oberg. 1999. Why the Mars Probe Went off Course. *IEEE Spectrum* 36, 12 (Dec. 1999), 34–39. DOI [10.1109/6.809121](https://doi.org/10.1109/6.809121) (Cited on page: 2)
- [205] Robert O’Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) (*USENIX ATC 2017*). USENIX Association, Berkeley, CA, USA, 377–389. ISBN 9781931971386 (Cited on page: 146)

- [206] Robert O’Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record And Replay For Deployability: Extended Technical Report. *CoRR* abs/1705.05937 (2017), 1–20. arXiv:1705.05937 <http://arxiv.org/abs/1705.05937> (Cited on page: 146)
- [207] Alessandro Orso and Bryan Kennedy. 2005. Selective Capture and Replay of Program Executions. In *Proceedings of the Third International Workshop on Dynamic Analysis* (St. Louis, Missouri) (*WODA 2005*). ACM, New York, NY, USA, 1–7. ISBN 1595931260 DOI [10.1145/1082983.1083251](https://doi.org/10.1145/1082983.1083251) (Cited on pages: 141 and 158)
- [208] Alessandro Orso and Gregg Rothermel. 2014. Software Testing: A Research Travelogue (2000–2014). In *FOSE 2014*. ACM, New York, NY, USA, 117–132. ISBN 9781450328654 DOI [10.1145/2593882.2593885](https://doi.org/10.1145/2593882.2593885) (Cited on pages: 10, 13, 103, 106, and 109)
- [209] Carlos Pacheco, Shuvendu K. Lahiri, Michael Dean Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*. IEEE Computer Society, Washington, DC, USA, 75–84. ISBN 0769528287 DOI [10.1109/ICSE.2007.37](https://doi.org/10.1109/ICSE.2007.37) (Cited on pages: 12, 70, 161, 180, and 182)
- [210] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the Validity of Mutation-Based Test Assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) (*ISSTA 2016*). Association for Computing Machinery, New York, NY, USA, 354–365. ISBN 9781450343909 DOI [10.1145/2931037.2931040](https://doi.org/10.1145/2931037.2931040) (Cited on page: 92)
- [211] Tanay Kanti Paul and Man Fai Lau. 2014. A Systematic Literature Review on Modified Condition and Decision Coverage. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing* (Gyeongju, Republic of Korea) (*SAC 2014*). ACM, New York, NY, USA, 1301–1308. ISBN 9781450324694 DOI [10.1145/2554850.2555004](https://doi.org/10.1145/2554850.2555004) (Cited on page: 103)
- [212] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding Myths and Realities of Test-Suite Evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) (*FSE 2012*). Association for Computing Machinery, New York, NY, USA, 1–11. ISBN 9781450316149 DOI [10.1145/2393596.2393634](https://doi.org/10.1145/2393596.2393634) (Cited on page: 72)
- [213] Paul Piwowarski, Mitsuru Ohba, and Joe Caruso. 1993. Coverage Measurement Experience During Function Test. In *Proceedings of the 15th International Conference on Software Engineering* (Baltimore, Maryland, USA) (*ICSE 1993*). IEEE Computer Society Press, Los Alamitos, CA, USA, 287–301. ISBN 0897915887 (Cited on page: 23)
- [214] Mounika Monugoti and Aleksandar Milenkovic. 2019. Enabling On-The-Fly Hardware Tracing of Data Reads in Multicores. *ACM Transactions in Embedded Computer Systems* 18, 4 (June 2019), 1–27. DOI [10.1145/3322642](https://doi.org/10.1145/3322642) (Cited on page: 30)
- [215] Md Tajmilur Rahman and Peter C. Rigby. 2018. The Impact of Failing, Flaky, and High Failure Tests on the Number of Crash Reports Associated with Firefox Builds. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) (*ESEC/FSE 2018*). ACM, New York, NY, USA, 857–862. ISBN 9781450355735 DOI [10.1145/3236024.3275529](https://doi.org/10.1145/3236024.3275529) (Cited on page: 139)
- [216] Paul Ralph and Ewan Tempero. 2018. Construct Validity in Software Engineering Research and Software Metrics. In *Proceedings of the 22Nd International Conference on Evaluation and Assessment in Software Engineering 2018* (Christchurch, New Zealand) (*EASE 2018*). ACM, New York, NY, USA, 13–23. ISBN 9781450364034 DOI [10.1145/3210459.3210461](https://doi.org/10.1145/3210459.3210461) (Cited on page: 57)
- [217] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. 2017. A Large-Scale Study of Programming Languages and Code Quality in GitHub. *Communications ACM* 60, 10 (Sept. 2017), 91–100. DOI [10.1145/3126905](https://doi.org/10.1145/3126905) (Cited on page: 79)
- [218] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A Large Scale Study of Programming Languages and Code Quality in Github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) (*FSE 2014*). Association for Computing Machinery, New York, NY, USA, 155–165. ISBN 9781450330565 DOI [10.1145/2635868.2635922](https://doi.org/10.1145/2635868.2635922) (Cited on page: 79)
- [219] Stuart C. Reid. 1997. An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing. In *Proceedings of the 4th International Symposium on Software Metrics (METRICS 1997)*. IEEE Computer Society, USA, 64–64. ISBN 0818680938 (Cited on page: 10)
- [220] Brian Robinson, Michael Dean Ernst, Jeff H. Perkins, Vinay Augustine, and Nuo Li. 2011. Scaling up Automated Test Generation: Automatically Generating Maintainable Regression Unit Tests for Programs.

- In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering* (Lawrence, KS, USA) (*ASE 2011*). IEEE Computer Society, USA, 23–32. ISBN 9781457716386 DOI [10.1109/ASE.2011.6100059](https://doi.org/10.1109/ASE.2011.6100059) (Cited on page: 143)
- [221] Ankit Rohatgi. 2020. *WebPlotDigitizer 4.2*. Retrieved 2020-01-10, archived by Internet Archive at <https://web.archive.org/web/20200110154215/https://automeris.io/WebPlotDigitizer/> from <https://automeris.io/WebPlotDigitizer> (Cited on page: 154)
- [222] Per Runeson. 2006. A Survey of Unit Testing Practices. *IEEE Software* 23, 4 (2006), 22–29. (Cited on page: 70)
- [223] Richard Rutledge, Sunjae Park, Haider Khan, Alessandro Orso, Milos Prvulovic, and Alenka Zajic. 2019. Zero-Overhead Path Prediction with Progressive Symbolic Execution. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) (*ICSE 2019*). IEEE Press, Piscataway, NJ, USA, 234–245. DOI [10.1109/ICSE.2019.00039](https://doi.org/10.1109/ICSE.2019.00039) (Cited on page: 30)
- [224] David Saff, Shay Artzi, Jeff H. Perkins, and Michael Dean Ernst. 2005. Automatic Test Factoring for Java. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (Long Beach, CA, USA) (*ASE 2005*). ACM, New York, NY, USA, 114–123. ISBN 1581139934 DOI [10.1145/1101908.1101927](https://doi.org/10.1145/1101908.1101927) (Cited on pages: 141, 143, and 158)
- [225] David Saff and Michael Dean Ernst. 2004. Automatic Mock Object Creation for Test Factoring. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Washington DC, USA) (*PASTE 2004*). Association for Computing Machinery, New York, NY, USA, 49–51. ISBN 1581139101 DOI [10.1145/996821.996838](https://doi.org/10.1145/996821.996838) (Cited on pages: 141 and 158)
- [226] Thomas Schulz, Lukasz Radliński, Thomas Gorges, and Wolfgang Rosenstiel. 2010. Defect Cost Flow Model: A Bayesian Network for Predicting Defect Correction Effort. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering* (Timisoara, Romania) (*PROMISE 2010*). Association for Computing Machinery, New York, NY, USA, 1–11. ISBN 9781450304047 DOI [10.1145/1868328.1868353](https://doi.org/10.1145/1868328.1868353) (Cited on page: 76)
- [227] Eric Shade. 2009. Size Matters: Lessons from a Broken Binary Search. *Journal of Computing Sciences in Colleges* 24, 5 (May 2009), 175–182. (Cited on pages: 53 and 54)
- [228] Sina Shamshiri, Rene Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE 2015)*. IEEE Computer Society, Washington, DC, USA, 201–211. ISBN 9781509000258 DOI [10.1109/ASE.2015.86](https://doi.org/10.1109/ASE.2015.86) (Cited on pages: 161 and 180)
- [229] Suchakrapani Datt Sharma and Michel Dagenais. 2016. Hardware-Assisted Instruction Profiling and Latency Detection. *The Journal of Engineering* 2016, 10 (2016), 367–376. DOI [10.1049/joe.2016.0127](https://doi.org/10.1049/joe.2016.0127) (Cited on page: 30)
- [230] Steven Skiena. 1991. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0201509431 (Cited on pages: 10 and 51)
- [231] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories* (St. Louis, Missouri) (*MSR 2005*). ACM, New York, NY, USA, 1–5. ISBN 1595931236 DOI [10.1145/1082983.1083147](https://doi.org/10.1145/1082983.1083147) (Cited on pages: 62 and 77)
- [232] Ben Smith and Laurie Ann Williams. 2008. *A Survey on Code Coverage as a Stopping Criterion for Unit Testing*. Technical Report. North Carolina State University, Department of Computer Science. (Cited on page: 56)
- [233] IEEE Computer Society, Pierre Bourque, and Richard E. Fairley. 2014. *Guide to the Software Engineering Body of Knowledge (SWEBOK)* (third ed.). IEEE Computer Society Press, Los Alamitos, CA, USA. ISBN 0769551661, 9780769551661 See also: ISO/IEC TR 19759:2005. (Cited on pages: 1, 2, 8, 9, and 16)
- [234] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. To Mock or Not to Mock? An Empirical Study on Mocking Practices. In *Proceedings of the 14th International Conference on Mining Software Repositories* (Buenos Aires, Argentina) (*MSR 2017*). IEEE Press, Piscataway, NJ, USA, 402–412. ISBN 9781538615447 DOI [10.1109/MSR.2017.61](https://doi.org/10.1109/MSR.2017.61) (Cited on pages: 162, 181, and 182)
- [235] Andreas Spillner, Tilo Linz, and Hans Schaefer. 2014. *Software Testing Foundations: A Study Guide for the Certified Tester Exam* (forth ed.). Rocky Nook, Santa Barbara, USA. ISBN 9781937538422 (Cited on page: 10)
- [236] Matt Staats, Gregory Gay, Michael Whalen, and Mats Heimdahl. 2012. On the Danger of Coverage Directed Test Case Generation. In *Proceedings of the 15th International Conference on Fundamental*



- Approaches to Software Engineering* (Tallinn, Estonia) (*FASE 2012*). Springer-Verlag, Berlin, Heidelberg, 409–424. ISBN 9783642288715 DOI [10.1007/978-3-642-28872-2\\\_28](https://doi.org/10.1007/978-3-642-28872-2\_28) (Cited on page: 103)
- [237] Richard Stallman, Roland Pesch, Stan Shebs, et al. 2019. *Debugging with GDB* (tenth ed.). Free Software Foundation, USA. (Cited on page: 149)
- [238] Richard Matthew Stallman and GCC Devevelopment Community. 2016. *GCC 7.0 Manual 1/2 (Volume 1)*. Samurai Media Limited, United Kingdom. ISBN 9789888406913, 9888406914 (Cited on page: 28)
- [239] David Ian Stern. 2011. From correlation to Granger Causality. *Crawford School Research Paper 1*, 13 (2011), 1–37. DOI [10.2139/ssrn.1959624](https://doi.org/10.2139/ssrn.1959624) (Cited on page: 90)
- [240] Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. 2010. *JUnit in Action* (second ed.). Manning Publications Co., Greenwich, CT, USA. ISBN 1935182021, 9781935182023 (Cited on page: 8)
- [241] Robert Endre Tarjan. 1985. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods* 6, 2 (1985), 306–318. (Cited on page: 51)
- [242] Gregory Tassej. 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing. *National Institute of Standards and Technology, RTI Project 7007*, 011 (2002), 429–489. (Cited on pages: 2 and 69)
- [243] Dávid Tengeri, Árpád Beszédes, Tamás Gergely, László Vidács, Dávid Havas, and Tibor Gyimóthy. 2015. Beyond Code Coverage – An Approach for Test Suite Assessment and Improvement. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, IEEE Computer Society, Washington, DC, USA, 1–7. DOI [10.1109/ICSTW.2015.7107476](https://doi.org/10.1109/ICSTW.2015.7107476) (Cited on page: 92)
- [244] Jörg Thalheim, Pramod Bhatotia, and Christof Fetzer. 2016. INSPECTOR: Data Provenance Using Intel Processor Trace (PT). In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, Washington, DC, USA, 25–34. DOI [10.1109/ICDCS.2016.86](https://doi.org/10.1109/ICDCS.2016.86) (Cited on page: 30)
- [245] The Linux Information Project. 2006. *Source Code Definition*. The Linux Information Project, USA. [http://www.linfo.org/source\\_code.html](http://www.linfo.org/source_code.html) (Cited on page: 24)
- [246] The Unicode Consortium. 2019. *The Unicode Standard 12.1*. Technical Report Unicode Standard 12.1. The Unicode Consortium, Mountain View, CA: The Unicode Consortium. ISBN 9781936213252 <https://www.unicode.org/versions/Unicode12.1.0/> Latest version: <http://www.unicode.org/versions/latest/>. (Cited on page: 24)
- [247] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. MSeqGen: Object-Oriented Unit-Test Generation via Mining Source Code. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Amsterdam, The Netherlands) (*ESEC/FSE 2009*). ACM, New York, NY, USA, 193–202. ISBN 9781605580012 DOI [10.1145/1595696.1595725](https://doi.org/10.1145/1595696.1595725) (Cited on pages: 161 and 180)
- [248] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2002. Efficient Instrumentation for Code Coverage Testing. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Roma, Italy) (*ISSTA 2002*). ACM, New York, NY, USA, 86–96. ISBN 1581135629 DOI [10.1145/566172.566186](https://doi.org/10.1145/566172.566186) (Cited on pages: 23 and 29)
- [249] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex: White Box Test Generation for .NET. In *Proceedings of the 2Nd International Conference on Tests and Proofs* (Prato, Italy) (*TAP 2008*). Springer-Verlag, Berlin, Heidelberg, 134–153. ISBN 9783540791232 (Cited on pages: 161, 180, and 182)
- [250] Undo. 2020. *UndoDB – An Interactive Reverse Debugger for C/C++*. Undo. Retrieved 2020-01-10, archived by Internet Archive at <https://web.archive.org/web/20200110153106/https://undo.io/solutions/products/live-recorder/> from <https://undo.io/products/undodb/> (Cited on pages: 143, 146, and 149)
- [251] Vijay V. Vazirani. 2001. *Approximation Algorithms*. Springer-Verlag, Berlin, Heidelberg. ISBN 3540653678 (Cited on page: 37)
- [252] Todd L. Veldhuizen. 2003. *C++ Templates are Turing Complete*. Technical Report. IndianaUniversity. (Cited on page: 175)
- [253] Ana-Maria Visan, Kapil Arya, Gene Cooperman, and Tyler Denniston. 2011. URDB: A Universal Reversible Debugger Based on Decomposing Debugging Histories. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems* (Cascais, Portugal) (*PLOS 2011*). ACM, New York, NY, USA, 1–5. ISBN 9781450309790 DOI [10.1145/2039239.2039251](https://doi.org/10.1145/2039239.2039251) (Cited on page: 146)
- [254] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. 2006. Time-Aware Test Suite Prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis* (*ISSTA 2006*). ACM, New York, NY, USA, 1–12. ISBN 9781595932631 DOI [10.1145/1146238.1146240](https://doi.org/10.1145/1146238.1146240)

- (Cited on pages: [107](#), [108](#), and [120](#))
- [255] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2011. Precise Identification of Problems for Structural Test Generation. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) (*ICSE 2011*). ACM, New York, NY, USA, 611–620. ISBN 9781450304450 DOI [10.1145/1985793.1985876](#) (Cited on pages: [180](#) and [182](#))
  - [256] Qian Yang, J. Jenny Li, and David Weiss. 2006. A Survey of Coverage Based Testing Tools. In *Proceedings of the 2006 International Workshop on Automation of Software Test* (Shanghai, China) (*AST 2006*). ACM, New York, NY, USA, 99–103. ISBN 1595934081 DOI [10.1145/1138929.1138949](#) (Cited on page: [23](#))
  - [257] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification and Reliability* 22, 2 (March 2012), 67–120. DOI [10.1002/stv.430](#) (Cited on pages: [13](#), [107](#), [118](#), [120](#), and [159](#))
  - [258] Hiroaki Yoshida, Guodong Li, Takuki Kamiya, Indradeep Ghosh, Sreeranga P. Rajan, Susumu Tokumoto, Kazuki Munakata, and Tadahiro Uehara. 2017. KLOVER: Automatic Test Generation for C and C++ Programs, Using Symbolic Execution. *IEEE Software* 34, 5 (2017), 30–37. DOI [10.1109/MS.2017.3571576](#) (Cited on pages: [151](#), [154](#), [155](#), [159](#), and [181](#))
  - [259] Hiroaki Yoshida, Susumu Tokumoto, Mukul R. Prasad, Indradeep Ghosh, and Tadahiro Uehara. 2016. FSX: A Tool for Fine-Grained Incremental Unit Test Generation for C/C++ Programs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) (*FSE 2016*). ACM, New York, NY, USA, 1052–1056. ISBN 9781450342186 DOI [10.1145/2950290.2983937](#) (Cited on page: [181](#))
  - [260] Hiroaki Yoshida, Susumu Tokumoto, Mukul R. Prasad, Indradeep Ghosh, and Tadahiro Uehara. 2016. FSX: Fine-Grained Incremental Unit Test Generation for C/C++ Programs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) (*ISSTA 2016*). ACM, New York, NY, USA, 106–117. ISBN 9781450343909 DOI [10.1145/2931037.2931055](#) (Cited on page: [181](#))
  - [261] Dongjiang You, Zhenyu Chen, Baowen Xu, Bin Luo, and Chen Zhang. 2011. An Empirical Study on the Effectiveness of Time-Aware Test Case Prioritization Techniques. In *Proceedings of the 2011 ACM Symposium on Applied Computing* (*SAC 2011*). ACM, New York, NY, USA, 1451–1456. ISBN 9781450301138 DOI [10.1145/1982185.1982497](#) (Cited on pages: [107](#), [108](#), and [120](#))
  - [262] Linbin Yu, Yu Lei, Raghu N. Kacker, and David Richard Kuhn. 2013. ACTS: A Combinatorial Test Generation Tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, IEEE Computer Society, Washington, DC, USA, 370–375. DOI [10.1109/ICST.2013.52](#) (Cited on pages: [99](#) and [101](#))
  - [263] Yanbing Yu, James A. Jones, and Mary Jean Harrold. 2008. An Empirical Study of the Effects of Test-Suite Reduction on Fault Localization. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) (*ICSE 2008*). ACM, New York, NY, USA, 201–210. ISBN 9781605580791 DOI [10.1145/1368088.1368116](#) (Cited on page: [127](#))
  - [264] Andy Zaidman, Bart Rompaey, Arie Deursen, and Serge Demeyer. 2011. Studying the Co-Evolution of Production and Test Code in Open Source and Industrial Developer Test Processes Through Repository Mining. *Empirical Software Engineering* 16, 3 (2011), 325–364. (Cited on page: [70](#))
  - [265] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. 2013. Bridging the Gap Between the Total and Additional Test-Case Prioritization Strategies. In *Proceedings of the 2013 International Conference on Software Engineering* (*ICSE 2013*). IEEE Press, Piscataway, NJ, USA, 192–201. ISBN 9781467330763 (Cited on page: [107](#))
  - [266] Lu Zhang, Shan-Shan Hou, Chao Guo, Tao Xie, and Hong Mei. 2009. Time-Aware Test-Case Prioritization Using Integer Linear Programming. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (*ISSTA 2009*). ACM, New York, NY, USA, 213–224. ISBN 9781605583389 DOI [10.1145/1572272.1572297](#) (Cited on page: [107](#))
  - [267] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael Dean Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) (*ISSTA 2014*). Association for Computing Machinery, New York, NY, USA, 385–396. ISBN 9781450326452 DOI [10.1145/2610384.2610404](#) (Cited on page: [140](#))
  - [268] Sai Zhang, David Saff, Yingyi Bu, and Michael Dean Ernst. 2011. Combined Static and Dynamic Automated Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) (*ISSTA 2011*). ACM, New York, NY, USA, 353–363. ISBN 9781450305624 DOI [10.1145/2001420.2001463](#) (Cited on pages: [161](#), [180](#), and [182](#))
  - [269] Luyin Zhao and Sebastian Elbaum. 2000. A Survey on Quality Related Activities in Open Source. *ACM*

- SIGSOFT Software Engineering Notes* 25, 3 (2000), 54–57. (Cited on page: [70](#))
- [270] Hong Zhu, Patrick Hall, and John May. 1997. Software Unit Test Coverage and Adequacy. *Comput. Surveys* 29, 4 (Dec. 1997), 366–427. (Cited on page: [56](#))
- [271] Thomas Zimmermann and Nachiappan Nagappan. 2008. Predicting Defects Using Network Analysis on Dependency Graphs. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) (*ICSE '08*). Association for Computing Machinery, New York, NY, USA, 531–540. ISBN 9781605580791 DOI [10.1145/1368088.1368161](https://doi.org/10.1145/1368088.1368161) (Cited on pages: [91](#) and [93](#))