TECHNISCHE
UNIVERSITÄT
DARMSTADT

Automated Quality-Assurance Techniques for the Modernization of Software-Product Lines

DEM FACHBEREICH ELEKTROTECHNIK UND INFORMATIONSTECHNIK
DER TECHNISCHEN UNIVERSITÄT DARMSTADT
ZUR ERLANGUNG DES AKADEMISCHEN GRADES
EINES DOKTOR-INGENIEURS (DR.-ING.)
GENEHMIGTE DISSERTATION

VON

SEBASTIAN RULAND

ERSTREFERENT: PROF. DR. RER. NAT. ANDY SCHÜRR
KORREFERENTIN: PROF. DR.-ING. INA SCHAEFER
KORREFERENT: PROF. DR. RER. NAT. HABIL. MALTE LOCHAU

TAG DER EINREICHUNG: 30.11.2021
TAG DER DISPUTATION: 16.02.2022

DARMSTADT 2022

Ruland, Sebastian
Automated Quality-Assurance Techniques for the Modernization of Software-Product Lines
Darmstadt, Technische Universität Darmstadt
Jahr der Veröffentlichung der Dissertation auf TUprints: 2022
URN: urn:nbn:de:tuda-tuprints-214888
Tag der mündlichen Prüfung: 16.02.2022

ACKNOWLEDGMENT

First, I would like to express my deepest gratitude to my supervisors Andy Schürr and Malte Lochau. Their support and collaboration have been extremely helpful for writing my thesis. I also want to thank my colleagues at the Real-Time Systems Lab and the Software Factory 4.0 project. Moreover, I want to thank every student who worked with me during my time as a PhD student under my supervision.

Additionally, I want to thank my family and friends, who supported me over the years and accepted the little time I had available when a paper was due.

Finally, and mostly, I want to thank my wife for her incredible support and her exceptional understanding. I could not have undertaken this journey without her.

Software-Factory 4.0

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

ERKLÄRUNG ZUR VERFASSUNG DER ARBEIT

§8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 30.11.2021

Sebastian Ruland

ABSTRACT

Nowadays, quality assurance for highly configurable software has become increasingly important. One of the most established ways to ensure software quality is testing. For example, old legacy systems are enhanced (called retrofitting) to cope with new Industrie 4.0 demands and the variability of the environments. During retrofitting, changes to the systems must not introduce new bugs, and backward compatibility must be ensured. Additionally, software is often implemented as a software-product line (i.e., a family of similar but distinguishable products) to include the variability. This introduces three challenges for quality assurance in Industrie 4.0. First, testing single and often safety-critical products. Second, ensuring backward compatibility and quality assurance during retrofitting and lastly, to cope with configurability during testing. To automate the generation of test cases for quality assurance, different tools exist. Those tools usually extract test goals for a given program based on a coverage criterion (e.g., statement coverage) and try to create test cases to cover all test goals. Additionally, in different environments, the focus on efficiency (e.g., CPU time) or effectiveness (the number of bugs found) differs. For safety-critical systems (e.g., autonomous systems in Industrie 4.0), software quality is crucial. Therefore, the focus is on effectiveness. In other cases (e.g., certification of systems) the focus is on efficiency due to fixed requirements for testing. Although effectiveness and efficiency often obstruct each other, both, and especially the trade-off between both can be improved. However, currently few studies are concerned with the trade-off during testing and the impact of different test-case generation strategies on the trade-off.

This thesis presents novel techniques to increase efficiency, effectiveness and optimize the trade-off for test-case generation. First, we focus on increasing efficiency for single products (e.g., for certification purposes). Since generating test cases for a single product often leads to many test goals, we developed new techniques for test-case generation with many test goals. First, we group test goals into partitions and execute a single reachability analysis for each partition. Second, we combine different analysis techniques to further increase efficiency. We evaluate different partitioning strategies and combine two different analysis techniques with different time limits to study the impact on efficiency.

Next, we present novel techniques to increase effectiveness during regression testing (i.e., testing changes, e.g., during retrofitting). The goal of regression testing is to detect bugs introduced due to the changes. However, as bugs are not known beforehand, no test goal exists that guarantees to detect the bug. Still, some test goals lead to test cases with high chances to detect a bug. For example, test goals resulting in test cases leading to different output behaviors compared to the previous program version are usually more effective compared to test cases only traversing the program changes. The methodology developed in this thesis enables us to configure novel test-case generation strategies (and test-suite reduction) to tune the resulting test suite in terms of effectiveness, efficiency, and especially in terms of the trade-off. In our evaluation, we also measured the impact of different parameters of our methodology on effectiveness and efficiency.

Lastly, to cope with the configurability of software-product lines, specific testing techniques are needed. One established technique to cope with the configurability during testing is sampling. Sampling derives a set of products from the product line and uses conventional testing techniques for each product. However, the effectiveness of the sample often remains unknown. To this end, we developed a technique based on mutation testing to measure the quality of sampling strategies based on effectiveness and efficiency. All techniques and methodologies developed during this thesis are motivated and explained on a running example and evaluated on real-world and synthetic programs.

KURZFASSUNG

Qualitätssicherung von hoch-konfigurierbaren Systemen wird heutzutage immer wichtiger. Das Testen ist eine der bewährtesten Methoden, um die Qualität solcher Software zu sichern. Zum Beispiel müssen Altsysteme, die verbessert werden (auch Retrofitting genannt), um mit neuen Ansprüchen im Industrie-4.0-Umfeld und dessen Variabilität umgehen zu können, getestet werden. Während solcher Verbesserung ist es essenziell, dass keine neuen Fehler eingefügt werden und die Systeme rückwärtskompatibel bleiben. Zusätzlich wird die Software oft als Software-Produktlinie (d. h. eine Familie ähnlicher, aber unterscheidbarer Produkte) implementiert, um die Variabilität zu unterstützen. Daraus ergeben sich drei Herausforderungen an die Qualitätssicherung im Industrie-4.0-Kontext. Die erste Herausforderung ist das Testen von einzelnen und oft sicherheitsrelevanten Produkten. Die zweite Herausforderung ist die Sicherstellung der Rückwärtskompatibilität und die Qualitätssicherung während des Retrofittings. Die letzte Herausforderung ist der Umgang mit der Konfigurierbarkeit von Systemen während des Testens. Um die Testfallgenerierung der Qualitätssicherung zu automatisieren, existieren verschiedene Tools. Diese Tools extrahieren für gewöhnlich zuerst Testziele aus einem gegebenen Programm, basierend auf einem Überdeckungskriterium (z. B. Anweisungsüberdeckung), und versuchen Testfälle zu erzeugen, die alle Testziele abdecken. Zusätzlich wird der Schwerpunkt in verschiedenen Branchen unterschiedlich auf die Effizienz (z. B. die CPU-Zeit) oder die Effektivität (die Anzahl der gefundenen Fehler) gelegt. Für sicherheitskritische Systeme (z. B. autonome Systeme in Industrie 4.0) ist die Softwarequalität essenziell. Daher liegt hier der Fokus auf Effektivität. In anderen Fällen (z. B. bei der Zertifizierung von Systemen) liegt der Fokus auf der Effizienz, da die Ansprüche an das Testen bereits vorgeschrieben sind. Auch wenn die Effektivität und die Effizienz sich oft gegenseitig behindern, können beide und vor allem der Trade-Off dennoch verbessert werden. Allerdings gibt es derzeit wenige Studien, die sich mit dem Trade-Off und der Auswirkung von verschiedenen Testfallgenerierungsstrategien befassen.

Diese Arbeit entwickelt neuartige Techniken, um die Effizienz, Effektivität und den Trade-off bei der Testfallgenerierung zu optimieren. Zuerst gehen wir auf neue Verbesserungen der Effizienz bei der Testfallgenerierung einzelner Produkte (z. B. für Zertifizierungen) ein. Da die Testfallgenerierung für einzelne Produkte oft viele Testziele beinhaltet, haben wir für dieses Szenario ein neues Verfahren entwickelt. Dazu gruppieren wir die Testziele zuerst in sogenannte Partitionen und führen für jede Partition eine Erreichbarkeitsanalyse durch. Zusätzlich kombinieren wir verschiedene Analysetechniken, um die Effizienz weiter zu steigern. Außerdem evaluieren wir verschiedene Partitionierungsstrategien und kombinieren diese mit zwei verschiedenen Analysetechniken mit unterschiedlichen Zeitlimits, um die Auswirkungen auf die Effizienz zu untersuchen.

Als Nächstes präsentieren wir eine neue Technik, um die Effektivität während des Regressionstestens zu erhöhen (d. h. dem Testen von Änderungen, z. B. während des Retrofittings). Das Ziel von Regressionstesten ist die Erkennung von Fehlern, die durch die Änderungen neu eingeführt wurden. Allerdings sind Fehler nicht vorher bekannt, daher gibt es keine Testziele, die die Detektion von Fehlern garantieren. Dennoch gibt es Testziele, die zu Testfällen führen, die mit hoher Wahrscheinlichkeit Fehler finden. Zum Beispiel gibt es Testziele, die zu Testfällen führen, die ein unterschiedliches Verhalten des neuen Programms im Gegensatz zum alten Programm aufzeigen. Diese Testfälle sind in der Regel effektiver im Vergleich zu Testfällen, die die Änderungsstellen nur erreichen. In dieser Arbeit haben wir eine Methode entwickelt, die es uns erlaubt eine Konfigurierung neuer

Testfallgenerierungsstrategien (und Testsuite Reduktion), um die resultierende Testsuite hinsichtlich ihrer Effektivität, Effizienz und dem Trade-off zu justieren.

Zuletzt, um die Konfigurierbarkeit von Software-Produktlinien zu bewältigen, sind bestimmte Techniken für das Testen notwendig. Eine sehr etablierte Technik ist das Sampling. Dazu wird eine Menge an Produkten aus der Produktlinie abgeleitet und diese dann mit konventionellen Testmethoden getestet. Allerdings ist die Effektivität eines Sampling-Verfahrens oft nicht bekannt. Aus diesem Grund haben wir eine Technik entwickelt, die basierend auf Mutationstesten die Qualität von samplebasierten Strategien hinsichtlich ihrer Effektivität und Effizienz misst. Alle Techniken und Methoden, die in dieser Arbeit entwickelt wurden, werden anhand eines Beispiels motiviert und erklärt sowie zusätzlich anhand von Echtwelt- und synthetischen Programmen evaluiert.

CONTENTS

ARG	**A**bstract **R**eachability **G**raph
AST	**A**bstract **S**yntax **T**ree
CAD	**C**omputer **A**ided **D**esign
CEGAR	**C**ounter**example** **G**uided **A**bstraction **R**efinement
CFA	**C**ontrol **F**low **A**utomaton
CNC	**C**omputerized **N**umerical **C**ontrol
CPS	**C**yber-**P**hysical **S**ystem
FQL	**F**Shell **Q**uery **L**anguage
ILP	**I**nteger **L**inear **P**rogramming
PL	**P**roduct **L**ine
SPL	**S**oftware **P**roduct **L**ine
SPLE	**S**oftware **P**roduct **L**ine **E**ngineering
SSA	**S**tatic **S**ingle **A**ssignment
SUT	**S**ystem **u**nder **T**est
CPU	**C**entral **P**rocessing **U**nit
SMT	**S**atisfiability **M**odulo **T**heories
IOT	**I**nternet **O**f **T**hings
CPS	**C**yber **P**hysical **S**ystem
OPC-UA	**O**pen **P**latform **C**ommunications - **U**nified **A**rchitecture
RIPR	**R**each, **I**nfect, **P**ropagate and **R**eveal
CPA	**C**onfigurable **P**rogram **A**nalysis
PC	**P**ersonal **C**omputer
SV	**S**oftware **V**erification
UML	**U**nified **M**odeling **L**anguage
OO	**O**bject **O**riented
NP	**N**ondeterministic **P**olynomial time
NRT	**N**umber of **R**egression **T**est cases
NPR	**N**umber of previous **P**rogram **R**evisions
RTC	**R**egression **T**est-case selection **C**riterion
RS	**R**eduction **S**trategy
CR	**C**ontinuous **R**eduction
API	**A**pplication **P**rogramming **I**nterface
MIP	**M**ixed-**I**nteger **L**inear **P**rogramming
MT	**M**utation **T**raversing
MR	**M**utation **R**evealing
HTTP	**H**ypertext **T**ransfer **P**rotocol
SIR	**S**oftware-artifact **I**nfrastructure **R**epository
EQ	**EQ**ivalent
NEQ	**N**on-**EQ**ivalent
PEQ	**P**artially-**EQ**uivalent
PL	**P**roduct **L**ine

1

INTRODUCTION

Nowadays, quality assurance for highly configurable software has become increasingly important. One of the most established ways to ensure software quality is testing. For example, in the current advent of Industrie 4.0, old legacy systems are enhanced (called retrofitting) to cope with the new demands. During retrofitting, changes to the systems must not introduce new bugs, and backward compatibility must be ensured. Additionally, highly configurable systems are used to cope with the variability of Industrie 4.0 environments. The software of those systems needs to support the configurability and, therefore, is often implemented as a software-product line (i.e., a family of similar but distinguishable products). In the remainder of this chapter, we will first provide an overview of the research areas covered in this thesis, followed by a discussion about the challenges tackled in this thesis.

The most prominent approach for developing highly-configurable systems is software-product-line engineering (SPLE). [144] The goal of SPLE is to design and develop a product family consisting of similar but distinguishable products. For example, a family of plotters consisting of a laser-plotter, a pen-plotter, etc. One of the two key design elements of SPLE is the development of a core functionality shared by all products of the family (e.g., the axis control of plotters). The second key design element is the variable functionality (e.g., plotting with a laser, or plotting with a pen), which is encapsulated in so-called features. Those features allow high-level customization options by selecting or deselecting these features (and, therefore, activating or de-activating the corresponding functionalities). For a given configuration, a product can then be derived, and, therefore, tailored products for individual customers can be easily created.

The development of SPLE is also split into two processes, namely domain engineering and application engineering. [144] The process domain engineering is concerned with the design of (re-usable) artifacts necessary for building the SPL (i.e., the features and the corresponding functionality). Therefore, domain engineering is responsible for the design of the product family (i.e., all products) and, therefore, which features are needed. Additionally, domain engineering is concerned with the implementation of the features, however, not with the implementation of the resulting products [14, 144]. The process application engineering is responsible for building specific products tailored for customer needs. To this end, configurations over features (by selecting or deselecting features) are created, and a corresponding product is derived. Lastly, application engineering is concerned with the validation and verification of derived products.

Additionally, SPLs are also further enhanced after the initial development. Those enhancements could be due to retrofitting or maintenance (bug fixes, etc.). Especially in Industrie 4.0, backward compatibility is crucial. For example, given a set of instruction for a plotter should provide the same resulting plot for both an old and a new version of the plotter software. While certain parts of the new code might have different behavior compared to the old code, as long as the resulting plot is the same, backward compatibility is ensured. Therefore, changes made to the software need to be tested as well. Testing those changes is called regression testing.

1.1 Challenges

Test-Case Generation for Products. One of the key activities in quality assurance in SPLs is testing. As described, single products are derived from the SPL during application engineering and need to be validated. In some cases, this also involves specific testing criteria, for example, to certify a system (e.g., DO-178C used to certified software used in airborne systems [77]). However, manual testing is costly and error-prone.

To this end, multiple steps within the testing process can be automated (e.g., execution of tests, etc.). Additionally, test-case generation can also be automated. To this end, multiple studies are concerned with automated test-case generation. However, the reachability of test goals (e.g., a line of code) is in general undecidable. Therefore, different heuristics and techniques try to optimize efficiency (in terms of CPU time).

One promising technology for test-case generation is software-model checking. [30, 83] The test goals (e.g., reachability of a line of code) are encoded as negated reachability problem (e.g., assume a line of code is not reachable). The software-model checker tries to prove the unreachability of the test goal. If the test goal is reachable, a counter-example is generated from which a test case can be extracted (see Sect. 2.3). [30, 83]

However, starting a reachability analysis for each test goal is expensive in terms of CPU time, especially in the case of hundreds or thousands of test goals (e.g., for branch coverage for large programs). For many test goals, parts of the reachability analysis could be reused (e.g., if they share a common path) and, therefore, re-computing the information of the previous reachability analyses is redundant. One technique to cope with this problem is multi-property checking. [15] Multi-property checking computes a single reachability analysis for all test goals to prevent redundant computations of similar paths. However, for software-model checking, there exist different techniques to abstract from unnecessary information during reachability analysis. For some test goals, the needed information differs. Therefore, running a single reachability analysis for all test goals might hinder abstraction and increase the CPU time.

Additionally, there exist different techniques for reachability analysis with software-model checking. The choice of analysis technique also affects the CPU time, as different techniques have different strengths and weaknesses. For example, a predicate analysis is quite powerful (i.e., is able to solve most reachability prob-

lems) and, often, able efficiently to handle loops in a program. However, it is also computational expensive due to expensive SMT-solver calls (to solve the predicate, e.g., $a < 2$ && $a > 0$). [24] Another technique (called explicit value analysis) is fast for paths depending on few variables. However, it has troubles with loops and paths consisting of many variable assignments. [24] Therefore, depending on the technique used, a test goal might be covered very fast or not. However, the choice often cannot be made for the entire program under test, but rather for single test goals or groups of test goals.

The challenge for test-case generation for single products is the optimization of efficiency in terms of CPU time by finding a trade-off between running a reachability analysis for each test goal and running a single reachability analysis for all test goals and by combining different analysis techniques to utilize their strengths.

Test-Case Generation for Program Versions. Another challenge for testing is regression testing. The goal of regression testing is to test changes made to an existing system. Those changes may be due to retrofitting, bug fixes, new features, or even removal of obsolete features and code. In any case, all changes need to be tested to ensure that no new bugs have been introduced. Additionally, backward compatibility often needs to be ensured by testing as well. However, it is often infeasible to run the whole test suite each time a change has been made since test suites for large systems tend to run for a large amount of time. Especially in Industrie 4.0 scenarios, tests are often executed on a physical machine. This additionally leads to material wear and increased maintenance costs. Therefore, only a fraction of the whole test suite can be executed. [181]

To this end, different techniques have been proposed to make testing more focused on changes to the program. [181] Test-suite reduction aims at removing obsolete test cases from the test suite. Test cases can become obsolete due to code removal or changes in the control flow of the program and, therefore, are unable to cover the corresponding test goal or other test cases also cover the same test goal due to the changes. Test-case selection tries to select test cases from a test-suite that are more likely to detect regression bugs (i.e., a new bug introduced by the new changes). For example, usually, only test cases are selected that at least reach a changed part of the code (called (modification-)traversing test cases). Otherwise, the test case will never be able to detect a regression bug. [70] A more complex test-case selection criterion is (modification-)revealing test cases (also called differential test cases). [70] Those test cases not only reach the changes but also lead to different output behavior compared to the previous program version. Note that test cases often still need to be executed to ensure backward compatibility. Modification-revealing test cases are guaranteed to lead to different output behavior, however, the different output might be intended.

Those test-case selection strategies can also be utilized for test-case generation by creating test cases fulfilling those criteria (e.g., creating modification-revealing test cases). Additionally, test-case selection and test-suite reduction can be used in combination during regression testing (e.g., first reducing the test-suite and, next, selecting test cases from the reduced test-suite). However, those techniques might

obstruct each other (e.g., removing test-cases that might be needed for test-case selection for later program versions).

Therefore, the challenge for test-case generation for program versions is to optimize the effectiveness and efficiency of regression testing. However, optimizing both effectiveness and efficiency is often infeasible, as both goals obstruct each other. Therefore, finding an optimal trade-off between effectiveness and efficiency is vital.

Testing of SPLs. Testing during application engineering of SPLs can be done on single products. However, testing all products is usually infeasible for real-world SPLs due to the exponential growth of products compared to the number of features. Especially in Industrie 4.0, testing multiple product might lead to manual labor to reconfigure the physical machine and, therefore, testing multiple products is even more expensive. To this end, several methodologies have been proposed to tackle this issue. One of the most promising methods is sample-based testing (also called sampling). [62] For sample-based testing, a representative set of products is chosen and tested. The goal is to choose the set of products that all (or most) bugs are present in at least one of those products and, therefore, are detectable during testing. However, as it is impossible to know the bugs beforehand, the optimal choice of a representative set of products is also unknown. To this end, different heuristics are used to compute a set of products for sampling. A widely used sampling criterion is τ-wise combinatorial feature-interaction coverage requiring every valid combination of selections and deselections of τ features to be contained in at least one test configuration $c \in S$. [63, 126] However, the number of products grows exponentially compared to the number of τ. Although, the probability of finding bugs might also increase by increasing τ.

The challenge for testing of SPLs is, therefore, to choose a representative set of products (e.g., by τ-wise sampling) to optimize the trade-off between efficiency (i.e., the number of products that need to be tested) and effectiveness (the number of bugs found).

1.2 Scientific Contributions

This thesis incorporates different testing strategies used for testing SPLs. To validate SPLs with software testing, this thesis uses automated test-case generation techniques to generate test-suites. One essential contribution is the optimization of efficiency and effectiveness for different testing scenarios. The efficiency depends on the testing scenario either focused on CPU time of the test-case generation ($efficiency_{CPU}$), the test-suite size ($efficiency_{size}$) for software products, or the number of products that need to be tested ($efficiency_{samplesize}$) for SPL testing. The efficiency metrics used in the following chapters are explained in detail in their corresponding chapters. The effectiveness is the percentage reached of a corresponding coverage criterion. That criterion is branch coverage in Chapter 3 and fault coverage in Chapter 4 and 5.

The scientific contributions of this work are concerned with the optimization of the automated generation of test-suites for either products or product-version histories and the testing of SPLs.

Contribution (1) - Efficient Test-Suite Generation for Software Products. For the first contribution of this thesis, we extend the test-case generation technique from Bürdek et al.[40]. The technique from Bürdek et al. uses multi-property checking to generate test cases for each test goal in a single reachability analysis. However, as mentioned before, trying to cover multiple test goals in a single reachability analysis might obstruct abstraction possibilities and, therefore, increase CPU time. To this end, the first extension is the grouping of test goals (called partitioning) and the execution of a reachability analysis per group of test goals (called a partition) with respect to different partitioning criteria. Partitioning allows the grouping of test goals to reuse information during reachability analysis and additionally prevents the loss of abstraction for tracking all test goals in a single reachability analysis. Additionally, partitioning allows for parallelization of the test-case generation since each reachability analysis for a partition can be run in parallel.

The second extension is by combining partitioning with the cooperative software-model checking from Beyer and Jakobs. [24] For this extension, we incorporate different analysis techniques in sequence to utilize their strengths and, therefore, further increase $\mathit{efficiency}_{CPU}$ during test-case generation.

Contribution (2) - Effective Test-Suite Generation for Software-Version Histories. For the second contribution of this thesis, we build a framework incorporating existing and novel techniques for generating test cases tailored for regression testing. To this end, we utilize the test-case generation technique from our first contribution to generate test cases that are either modification-traversing or modification-revealing (depending on the configuration of the framework). As stated before, execution of test cases might be necessary since the different behavior might be intended. However, might be costly (especially, if they need to be executed on a physical machine). To this end, we support different test-suite reduction strategies to increase $\mathit{efficiency}_{size}$ during regression testing.

We also developed a novel technique to create multiple test cases for a single test goal to increase effectiveness. Those test cases are guaranteed to traverse different program paths when executed. This framework allows to increase the effectiveness of regression testing and, additionally, to increase $\mathit{efficiency}_{CPU}$ or fine-tune the trade-off between effectiveness and $\mathit{efficiency}_{CPU}$and the trade-off between effectiveness and $\mathit{efficiency}_{size}$by configuring different parameters.

Contribution (3) - Evaluating SPL Testing Strategies. For the third contribution of this thesis, we build another framework to evaluate different testing strategies for SPLs. As mentioned before, testing each product of an SPL is usually infeasible. To this end, different techniques have been proposed to compute a sample consisting of products that should be tested. However, if the sample is effective in terms of bug detection is often an open question. To this end, our

framework incorporates mutation testing techniques for SPLs to introduce bugs in an SPL. Next, we use family-based test-case generation to check on which products the mutants are detectable on (i.e., for which product exists an input-vector leading to different output behavior for the mutated product and the original product). This information is then used to evaluate samples by calculating the number of mutants that can be detected on products of the given sample. Therefore, we are able to evaluate the effectiveness of different techniques proposed in the literature for sample-based SPL testing. Additionally, the $\mathit{efficiency}_{samplesize}$ is calculated and to measure the trade-off between effectiveness and $\mathit{efficiency}_{samplesize}$.

1.3 Outline

In Chapter 2, we will first introduce a motivational running example from Industrie 4.0 and a miniature example to further motivate and explain the concepts of this thesis. Additionally, we will explain the background information needed for the following chapters. In Chapter 3, we will provide additional background for test-case generation with software-model checking. Additionally, we introduce novel techniques to increase $\mathit{efficiency}_{CPU}$ for test-case generation of software products. In Chapter 4, we will provide additional background information for regression testing and explain our novel framework for increasing the effectiveness of regression testing. In Chapter 5, we will provide additional background information for sample-based SPL testing and introduce our framework for evaluating the effectiveness of sample-based SPL testing techniques in terms of bug detection. Finally, we will summarize our work in Chapter 6 and provide ideas for future work.

1.4 Previously Published Material

This thesis includes ideas and material that has been previously published in conferences and journals. Table 1.1 shows the publications from which content has been used for this thesis. A list with all scientific publications of the author of this thesis is found in Appendix C.

Table 1.1: List of Previously Published Material

Chapters	Publications
Chapter 3: Test-Case Generation for Software Products	Ruland et al. [156] Ruland et al. [157]
Chapter 4: Regression Testing	Ruland and Lochau [154]
Chapter 5: Testing of Software-Product Lines	Ruland et al. [155]

2

BACKGROUND

This chapter introduces a running example from the Industrie 4.0 domain to motivate the challenges tackled throughout this thesis. Next, we introduce a small program written in C to illustrate concepts and notations (see Sect. 2.1.3). Thereafter, we will explain basic concepts of software testing (see Sect. 2.2). In section 2.3, we show different concepts and techniques for automated test-case generation. Next, we describe regression testing concepts, such as test-suite optimization as well as special coverage criteria for regression testing. Lastly, we explain basic concepts of software-product lines and software-product line testing.

2.1 Industrie 4.0

During the last 17 decades, the industrial world has gone through three different industrial revolutions. The first industrial revolution, in the second half of the 18. century, was based on mechanical inventions and changes in production [20]. Later, the second industrial revolution took place, mainly focusing on mass production due to chemical and electrical advances [20]. The last industrial revolution is based on computer science and the advent of computer systems [20].

However, the industry is currently advancing towards the fourth industrial revolution, often called Industrie 4.0 or the Internet of Things (IoT)[175, 10]. Industrie 4.0 is based on intelligent, cyber-physical systems (CPS). These CPS are comprised of multiple intelligent systems, storage systems, production facilities, and more. These systems within a common CPS can autonomously interact, transfer data and trigger different actions or directly control each other. This enables new applications and business models, which contain autonomous systems to further optimize production. However, there are still many legacy systems comprised of machines incapable of connecting with cyber-physical systems in industrial plants. These legacy systems are rarely equipped with the necessary hardware and interfaces. [160]

To fully upgrade the production to cyber-physical systems, these legacy systems need to be either replaced or upgraded to be fully integrated into the CPS. Since the replacement of these machines is often quite costly (and in some scenarios not possible), different strategies to upgrade legacy systems have been subject to research in the last years. [160, 120, 122, 121]

In this thesis, we use a legacy system as motivation, which has been upgraded with certain functionalities to fit into an Industrie 4.0 scenario. The focus in this thesis is the software of the machine, which has to be upgraded as well to cope with the newly implemented hardware.

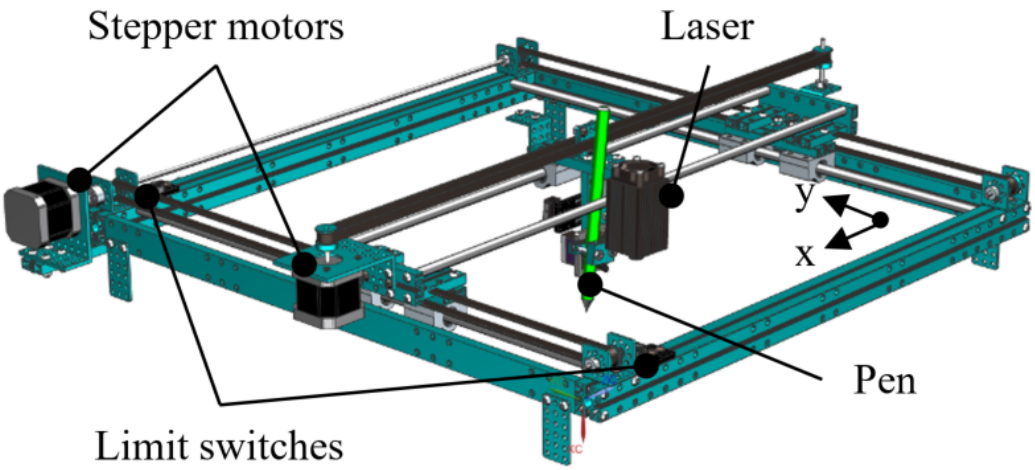


Figure 2.1: Laserplotter [113]

2.1.1 *Plotter*

We use a plotting machine as a running example for a legacy system. This machine was built and upgraded as part of the Industrie 4.0 cluster within the Software Factory 4.0 project[1]. The machine was upgraded to cope with new requirements due to a new environment, namely Industrie 4.0. The plotter is a common computerized numerical control (CNC-)machine, which was only able to print on paper with either a laser or a pen. Figure 2.1 shows the original version of the plotter [113]. It has two axes, which are moved by stepper motors. Additionally, the laser can either be turned on or off, and the pen can be lowered or pulled up. Lastly, on each axis is a limit switch, which is responsible for boundary checking (i.e., to trigger a function if the motors hit the boundary).

As control software grbl[2] is used. grbl is a CNC software that is quite popular in the maker scene. Grbl takes single lines of so-called G-code as input. G-code is a domain-specific language mainly developed for CNC machines. The language itself is quite simple. Most of the time, each line of G-code is exactly one command. While the language does support some control-flow structures, they are usually not used. The reason is, most of the time, G-code is automatically generated based on CAD models. Therefore, control structures are often not needed.

An example for a single G-code command is *G01 X5*, which states that the machine should move in a straight line (G01) to the coordinate 5 at the X-axis (X5).

Legacy Upgprades. To be able to participate in an Industrie 4.0 environment, different kinds of functionality should be provided by the machine. First, the machine needs to be able to connect to other machines. However, most legacy systems are not equipped with the needed interfaces. Therefore, their connectivity often has to be upgraded to enable information transfer and communi-

[1] https://www.software-factory-4-0.de
[2] https://github.com/grbl/grbl

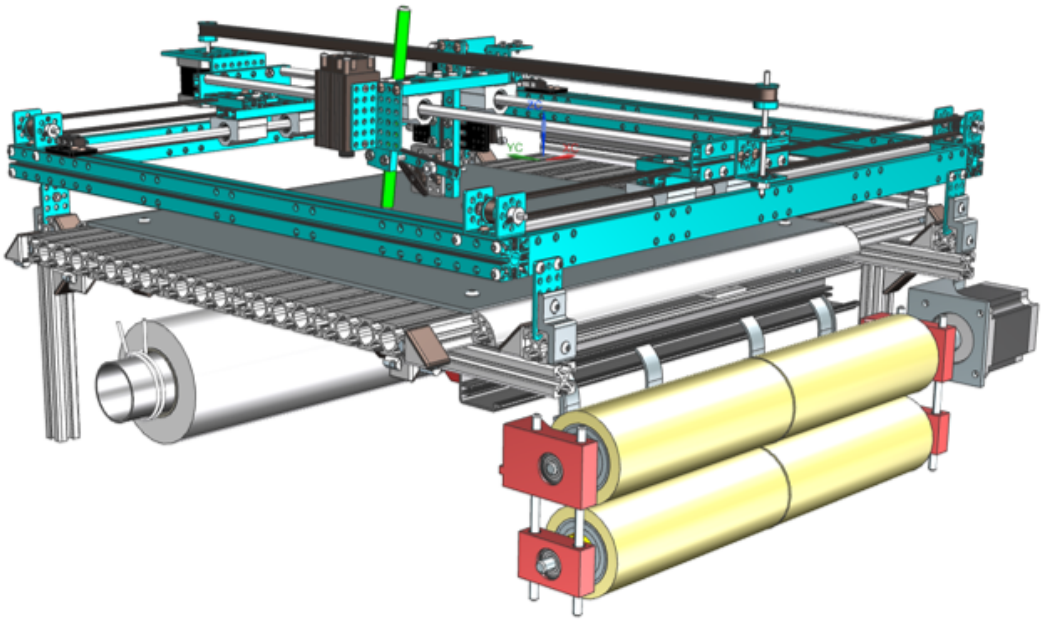


Figure 2.2: Laserplotter with Paper Feed [113]

cation with other machines in a common CPS. To provide the necessary interfaces, different solutions have been proposed in the last years. [38] One prominent example in science is OPC-UA, which is a machine-to-machine communication protocol [116, 148]. Second, user interaction with the machine should be minimized [150]. Since these systems should work autonomously, each user interaction might slow down the whole production process. Additionally, to work fully autonomously, the needed amount of information by specialists should be reduced to a minimum. Lastly, to maximize reuse additional functionality should be select-able by configuration. This enables the usage of different variants of a machine, without the need to implement each machine separately. This is called (software-) product lines (see sect. 2.5).

First Upgrade. The initial plotter was only able to plot on a single sheet of paper. After each plotting process, the sheet of paper had to be manually replaced. To minimize user interaction in this process, the system was equipped with a paper-feed mechanism.

For this purpose, a paper-roll and a motor were added to the machine. Figure 2.2 shows the additional hardware [113]. To control the additional motor, existing parts of the software have been re-used and extended. The new motor is also controllable by G-Code commands. By implementing this upgrade as a new variant of the plotter, we lifted our demonstrator to a product-line. This enables easy installation of a plotter with or without paper feed.

Second Upgrade. The second upgrade of the plotter is the implementation of a dry-run. Since grbl reads the G-Code statement-wise and executes each statement directly, an incorrect G-Code file might crash the plotting-job midway (e.g.,

incorrect bounds specified). Especially in an autonomous environment with potentially many similar machines, it is essential to know which machine is able to perform which job. For example, there could be multiple plotters with different page sizes (e.g., DIN A4 or DIN A3). These boundaries (or other constraints) might not be known a priori and can be evaluated by performing a dry-run. Additionally, this upgrade can be used in combination with a digital twin (i.e., a digital representation of the machine) to show the simulated movement of the plotter [134, 176].

Third Upgrade. The third planned update is adding communication possibilities to the plotter. To this end, OPC-UA is utilized to implement an asset administration shell [38]. This enables other OPC-UA components in the same network to find the plotter via OPC-UA, to receive information about the plotter and even to send data and commands.

By implementing these upgrades, the plotter is able to participate in an autonomous Industrie 4.0 scenario as it is able to work without much user interaction due to the first upgrade. It is able to test G-Codes before execution due to the second upgrade, and it can communicate via OPC-UA due to the third upgrade.

2.1.2 *Error-Codes*

Incorrect inputs to grbl, such as invalid G-Code commands or commands which violate constraints (e.g., leaving the axis boundaries), are reported via error-codes. These error-codes are essential for dealing with unwanted behavior.[146] Changes to grbl can modify behavior of already implemented G-Codes as well as introduce support for new G-Codes. In both cases backward compatibility is essential, as previously working G-Code commands still need to function properly. Especially, the parts of the source code responsible for error-handling (i.e., the grbl error-codes) have to be tested thoroughly to prevent incorrect behavior (e.g., not triggering an error-code due to incorrectly added features).

2.1.3 *Running Example.*

Since the full source code of grbl is too large to show in this thesis, we will use an additional, smaller running example to illustrate concepts and ideas. The example is written in C as well and serves as a representation for source code from grbl. Figure 2.3 shows the source code of the running example. This source code is the initial version P_0 of the example and still contains three bugs. These bugs are intentionally placed and fixed later on in this thesis to further illustrate concepts in the following chapters

Specification. The specification of the function (i.e., the description, not necessarily conforming to the implementation due to bugs) is as follows. The function **`find_last`** receives as inputs an integer-array **`x[]`** and an integer value **`y`**. The purpose of the function is to find and return the index of the last element of **`x[]`** with the same value as **`y`**. The first element of **`x[]`** carries as meta-information

```
1  int find_last (int x[], int y) {
2    if(x[0] <= 1)
3      return -1;
4
5    int last = -2;
6    for (int i=0; i <= x[0]-2; i++)
7      if (x[i] <= y)
8        last = i;
9    return last;
10 }
```

Figure 2.3: Initial Version P_0 of a Program Unit [154]

the length of the array. Therefore, the first element should be ignored during the search for the value of **y**. Additionally, the function returns error codes in specific cases. If the array is empty (i.e., only carries the meta-information **size**, which is smaller or equal to one), the function returns the error-code -1. Lastly, if the array does not contain any element with the same value as **y**, the function returns the error-code -2.

Bugs. As mentioned, there are currently three bugs in our implementation.

1. Bug B_1: The search index starts at 0, including the meta-information in the search (see line 6).
2. Bug B_2: The search index stops at $x[0] - 2$, which excludes the last element from the search (see line 6).
3. Bug B_3: The search matches all values smaller or equal to **y** instead of only equal values (see line 7).

These Bugs will persist for now and will be fixed later on to further illustrate concepts in the following chapters.

2.2 Software Testing

One of the key activities in software engineering is testing of the system. Bug fixing is increasingly expensive and difficult the longer the bug remains undetected [11]. Especially in safety-critical environments, bugs can lead to serious consequences. Therefore, it is imperative to find and fix bugs as soon as possible. Testing can be used to reveal bugs in the system. However, it is impossible to prove the absence of further bugs. To prove correct behavior, the system would need to be verified (i.e., to show for all possible inputs the correct behavior) and not only validated/tested (i.e., to show the correct behavior for a fixed amount of inputs). As testing all possible input values is often infeasible (even for small systems) for the sheer amount of test cases needed, testing tries to focus on a selected amount of properties that should be validated. Therefore, to test a system systematically and thoroughly, the testing should be split into different parts. In the following, we will explain different notions to specify which parts of the system are subject to a specific testing methodology.

There are two main properties of a software system, namely functional and non-functional properties.

- Functional properties describe the fundamental behavior of the System. This mainly contains the input and output behavior. For example, in our running example, the specification states that the function should return the error-code -1 if the array passed as parameter x is empty (i.e., only consists of the meta-information **size**). If we execute the function with the input values of $x = [0], y = 0]$ the function indeed returns -1. This means the functional property concerning this specific part of the specification is correctly implemented.
- Non-functional properties describe the remaining behavior of the system. Therefore, the focus on these tests is not the specified input/output behavior but rather the system's quality. This includes temporal behavior (e.g., how long does the system execute) or reliability of the system.

In this thesis, we focus on the functional properties of a system. Non-functional properties are out-of-scope.

2.2.1 *General Testing Process*

In general, the testing process can be divided into different levels [11]. This enables the tester to write different tests (in different granularities) for different parts of the system under test (SUT).

Unit Testing. The first level of the test process is the so-called unit test. Unit testing comprises testing of single components of the SUT (e.g., single functions, such as the running example shown in fig. 2.3). These units are tested independently of other units, as the focus is on the unit's internal correctness. To test these functions, usually, an input vector and an expected output value are used. [166]

> **Example 2.1.** In regards of the running example (see fig. 2.3) a single test case (i.e., an input vector and an optional output value) could be $t = ([0], 0), -1$. This test case has the input vector $([0], 0)$, meaning the value $[0]$ is passed to the first parameter as an argument, and the second value 0 is passed to the second parameter. Lastly, the value -1 is the expected output of the function.

In some cases, the output value is irrelevant (e.g., testing for crashes only needs to validate whether the function crashes or not). In this case, the expected output value can be omitted.

Integration Testing. The next level of the test process is the integration test. This testing step tests multiple components and their interaction. Therefore, the focus of integration testing are interfaces and interaction between components. For this purpose, multiple components are integrated into a subsystem. There are different integration strategies, such as big-bang, top-down, or bottom-up integration. [166]

System Testing. The third level of the test process is the system test. System testing is concerned with the system as a whole. The units have to be implemented and integrated, and a testing environment has to be built (similar to the runtime environment). The purpose of system testing is to test whether the system fulfills all business requirements. For this purpose, the system is tested from the user's point of view.[166]

Acceptance Test. The last level is the acceptance test. In this phase, the user/customer tests whether the software performs as intended (i.e., the user/-customer tests if the software is acceptable). Usually, this is the first time the user is involved in any of the testing processes.[166]

The methods introduced in this thesis are mainly concerned with unit testing. Although, some cases additionally test multiple units in a single step.

2.2.2 *WhiteBox- vs. BlackBox-Testing*

While test levels are concerned with the part of the SUT that should be tested, there are different notions on how to test these parts. In this chapter, we will introduce the notions of BlackBox-, WhiteBox- and GreyBox-testing. [166]

BlackBox Testing. During BlackBox testing, the system is considered as a black box, meaning the content of the system is unknown. BlackBox testing uses the specification of the system to validate the properties of the SUT. To fully test a SUT, a test case for each possible combination of input values is needed. Due to the sheer amount of test cases needed, this is most of the time infeasible for real systems. Therefore, test cases have to be selected to validate as much of the specification as possible while still being limited in the number of test cases. [166]

> **Example 2.2.** In our running example (see fig. 2.3), the specification states that the index of the last element with the same value as y is returned. This can be validated with a test case, such as $t = \{([2,1],1),1\}$. However, due to the bugs, the incorrect result of 0 will be returned. Therefore, this test can be created based on the specification and is able to reveal that at least one bug is present when executed.

As it is often difficult to create test cases which validate the SUT without any guidelines, different BlackBox testing strategies have been subject to research. [166] One prominent method is the so-called classification tree method. This strategy divides the input values into different classes and tries to merge values with the expectedly same behavior. These classes are called equivalence classes, as the behavior of all values in an equivalence class is supposed to behave equivalent when passed as parameter. Afterwards only one value of the given classes has to be selected as a test case. Sometimes, the boundary values between the equivalence classes are tested as well. [166] The most obvious advantage of BlackBox testing is the fact, that testing is performed from the user's perspective. Therefore, test cases often resemble inputs that a user might perform. Additionally, as the inner structure of the code is not necessary, BlackBlox testing can also be performed on

programs where the source code is unavailable (e.g., third-party libraries). However, in case of complex input values (e.g., arrays of indefinite length) finding relevant test cases is often hard. Additionally, finding test cases which test more complex parts of the code is often difficult. Lastly, there exist different requirements a test suite must fulfill (e.g., standards) which are hard to fulfill without knowledge about the source code.

WhiteBox Testing. WhiteBox testing primarily focuses on the internal representation (e.g., source code) of the system instead of the specification to generate test cases. WhiteBox testing aims to choose an input vector to exercise certain paths in the system (for example, to reach a certain line of code).[166]

> **Example 2.3.** In our running example, a test goal might be to reach line 9. To this end, we can check the source code to see that the test case needs to provide the value greater than 1 for the first element of the array (i.e., the size meta-information) to reach this line. The values for the other parameters do not matter. Therefore, a test case for this goal could be $t = \{([2,1],0),-\}$.

Since the internal structure of the code is known, it is easier to create test cases reaching specific parts of the code. Therefore, it is easier to reach complex parts of the code or even vital parts, such as the error-handling routines in grbl (or other Industrie 4.0 applications). Additionally, some requirements for test suites (e.g., standards, such as the RTCA/DO-178C standard) require specific parts of the code to be covered (e.g., 70% of all branches). For these cases WhiteBox testing is preferable to BlackBox testing. However, as the test cases are created based on the internal structure of the code and not the specification, the test cases often do not represent user behavior. Additionally, test case generation is often more expensive (e.g., in terms of working hours) compared to BlackBox testing. [166]

GreyBox Testing. A combination of BlackBox and WhiteBox testing is GreyBox testing. GreyBox testing uses the specification to create test cases. Additionally, it also takes part of the system's internal representation into account (e.g., branch conditions or loops). [166]

WhiteBox testing enables us to test specific internal properties of the source code itself. Especially in Industrie 4.0 this is vital (e.g., for testing proper error-code behavior and other error-handling routines). Therefore, the methods used in this thesis focus on WhiteBox testing and partly on GreyBox testing.

2.2.3 *Coverage Criteria*

The selection of test cases is often made based on so-called coverage criteria. There exist many different coverage criteria, with different advantages and disadvantages. Two of the most common coverage criteria are statement and branch coverage. [166] To fulfill statement coverage for each statement in the source code, the test suite must have at least one test case that will reach the statement when executed. To fulfill branch coverage for each branch in the source code, the test suite must have at least one test case that will reach the if-branch and the else-branch

when executed. In case a test suite fulfills branch coverage, the test suite also fulfills statement coverage.

Example 2.4. To fulfill statement coverage for our running example, all lines have to be executed at least once, and the loop in line 6 has to be executed twice in a single run to execute the statement $i++$. To this end, a test suite with the test cases $t = \{([1], 0), -\}$ and $t = \{([2, 1], 1), -\}$ can be used.

Branch coverage also requires each branch to be executed at least once (i.e., each condition needs to be evaluated to true and to false at least once). If we use the same test suite as for statement coverage, we still miss the execution of the "else"-branch of line 7. Therefore, we would need an additional test case, such as $t = \{([2, 1]0), -\}$.

For industrial software branch-coverage is often used as coverage criterion, as branch-coverage is often mandatory for safety-relevant source code (e.g., RTCA/-DO-178C standard).

However, fully satisfying a coverage criterion is often unrealistic (or even impossible). Therefore, a coverage criterion is often accompanied by a value, stating to which degree the coverage criterion needs to be satisfied (e.g., to satisfy statement coverage to 70%, at least 70% of all statements have to be executed at least once by the test suite). [166]

Additionally, coverage criteria often do not take interaction between different statements into account (e.g., a bug that only occurs when executing specific statements in a specific order is very hard to find with coverage driven testing).

As mentioned before, the goal of testing is to find as many bugs as possible early on. However, it is hard to link coverage criteria like branch or statement coverage to a "bug-finding potential". Therefore, research has tried to develop new coverage criteria that provide a more realistic interpretation of a bug finding potential of a test suite. [12]

2.2.4 *Mutation Based Testing*

To correctly evaluate the bug-finding potential of a test suite, real bugs would be needed. Additionally, since a test suite should find bugs that are unknown beforehand, we would actually need knowledge about future bugs. Since this is obviously impossible and even knowledge of old bugs are often not available in the amount needed to evaluate a test suite (as they are often not stored), research has tried to develop alternative ways to evaluate test suites. For this purpose, mutation testing has been developed. [135]

Mutation testing is based on two hypotheses [135]. First, the competent programmer hypothesis, which states that senior programmers seldom make cognitive mistakes but rather small syntactical faults [56]. The second hypothesis is the so-called coupling effect, which states that larger bugs can be broken down into small bugs that cascaded into the larger bug [56, 135].

Based on these hypotheses, mutation testing automatically makes syntactic changes to the source code (i.e., mutates it) and checks whether the test suite

is able to detect these changes (i.e., by having a test case that expects a different output value than returned by the mutated program). [2]

To detect a mutation, the test suite needs at least one test case, which fulfills the so-called *RIPR* criterion [11].

- **R**each: The test case has to reach the modification.
- **I**nfect: The test case has to infect the current program state (i.e., change the program state in comparison to the original program).
- **P**ropagate: The infected state must be propagated.
- **R**eveal: The propagated infection must be revealed.

If a test case manages to reach the modification, it is called *traversing* test case. If the test case additionally manages to infect, propagate and reveal the modification, it is called *revealing*.

> **Example 2.5.** An examplary mutation for our running example is to replace the statement *return* -1; with *return* 1; in line 3 (i.e., to remove the sign of the return value). This simulates a fault made by a programmer, who forgot the sign of the return value. To detect this mutation the test case $t = \{([0], 0), -1\}$ can be used, since it expects the return value of -1. If executed on the mutated code, the return value will be 1, and therefore, the test will fail and detect the mutation.

There are different catalogs with different syntactic changes within the context of mutation testing, which can be automatically introduced in the source code [2, 48, 162]. This way, many bugs can be created and used to evaluate a test suite without the knowledge of future bugs or the bug history of the system. The downside of mutation testing is the (CPU) time needed as, in the worst case scenario, all test cases have to be executed for each mutant. However, in safety critical scenarios, as it often is in Industrie 4.0, these additional costs often provide a decent trade-off in terms of higher effectiveness of test-suites. Additionally, the test cases are often executed on the machine itself. Since test suites for mutation detection are often smaller in size, the time needed to run the test cases on the machine is often less compared to coverage based test suites.

2.3 Test-Case Generation

Manual generation of test cases is often expensive and error-prone. To this end, different methods have been created to automatize test-case generation. In the following, we will explain how software model checking can be utilized to create test cases in an automated fashion.

2.3.1 *Software Model Checking*

Software model checking is a software analysis method used to prove or disprove certain properties of a system. The input for a software model checker is

the source code, which is then transformed into an internal representation. On this representation, the software model checker will verify or disprove a property based on a specification. Afterwards, the software model checker will either return a proof if the specification holds or a counter-example otherwise. This counterexample can be used to extract input values for the SUT and, therefore, create a test case for the disproved property. [31] This process can be used for automated test-case generation by specifying test goals (e.g., to reach a line of code) as negated properties (e.g., assume the line of code is not reachable by any input vector). Then, the software model checker will try to verify or disprove the property. If the property holds, the test goal is infeasible (e.g., the line of code is unreachable). Otherwise, a counter-example will be returned from which a test case can be extracted. [30]

2.3.1.1 *Control-Flow Automaton (CFA)*

A commonly used internal representation for programs is a so-called control-flow automaton (CFA). A control-flow automaton is a directed graph consisting of nodes and edges. Nodes represent program locations while edges represent operation (e.g., conditions or assignments). Additionally, there exists exactly one initial node without entering edges and exactly one final node without outgoing edges for each CFA.

> **Example 2.6.** Figure 2.4 shows the control-flow automaton of our running example in fig. 2.3. Node 1 is the initial node and represents the function entry. The first edge from node 1 to node 2 is the edge responsible for the assignment of the first parameter x. Node 3 is the first branching node representing the branching of the if-statement in line 2. Branching nodes always have two outgoing edges representing the "true" and "false" paths of the condition (i.e., the if- and else-branches) therefore, if the first element of x is larger or equal to 1 while in node 3, the right path will be taken to go to node 5. Lastly, nodes 4 and 8 are the nodes returning a value by taking the edge to node 0, the terminal node of the function.

2.3.1.2 *Abstract Reachability Graph (ARG)*

The control flow automaton can then be used for reachability analyses. This can be done by constructing a so-called reachability graph. A reachability graph has nodes representing all possible program states and can be used to check for reachability properties. [33] However, program states are often huge or even infinite (e.g., infinite loops). Therefore, it helps to abstract from concrete states to abstract states. Symbolic model checkers often build abstract reachability graphs (ARGs) to check for reachability properties.

> **Example 2.7.** An extract of an exemplary ARG for our running example is depicted in figure 2.5. Each abstract state consists of two sub-states. First, the state's location corresponds to the number of the node in the CFA (Fig. 2.4).

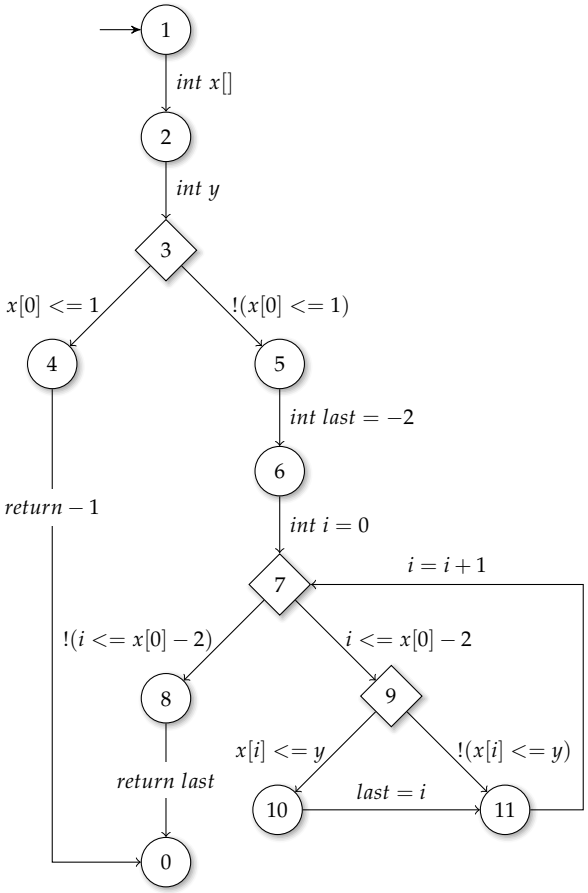


Figure 2.4: CFA of the C-Function FIND_LAST

The second box is the path condition (i.e., which formula needs to be adhered to to reach this abstract state). In terms of the first abstract state ($1|true$, or 1 for short), the location is node 1 in the CFA, and the path condition is *true*, as this state is reached independently of variable and parameter values. The transitions from one state to the next are always the edges of the CFA, such as $int\ x[]$ from state 1 to 2. Since there is no restriction to reach the next state, the path condition remains *true*. When traversing the edge from state 3 to 4 the path condition changes to $x[0]_0 <= 1$. Note the subscript at $x[0]_0$. This is used for single static assignments (SSA) [149]. Within the ARG, each variable is only assigned once. For the second assignment, the subscript (i.e., the SSA-value) is incremented. Therefore, x_i represents the value of x after the $i-th$ assignment. This enables us to build path conditions without loss of information by overriding values.

State 0 after node 4 is a final node. The value of -1 is assigned to a temporary return variable (*rTemp*), and the ARG stops at this node. When traversing from node 3 to 5, the same applies for the transition from 3 to 4. The path condition in node 6 is updated with the addition of $\wedge last = -2$ as the assignment of the variable *last* happens within the transition. The node $0'$ after node 8 has the same location as node 0 after node 4. However, as the path condition differs, we chose the location $0'$ as it is easier to refer to this specific node later on. Lastly, nodes $7'$ and $7''$ after the nodes 11 and $11'$ illustrate that after iterating through the loop once, the path condition expanded, and we are back at location 7. This step continues until the loop cannot be iterated anymore (e.g., due to the loop condition) or the reachability algorithm terminates (e.g., due to resource constraints). This shows that even for abstract states, the number of states in an ARG might be infinite depending on loops and jump-statements of the program.

This automated test case generation method can be used to create very specific test cases, at least as long as they are encoded as reachability problems. Additionally, this method is able to guarantee the non-reachability of test goals (as long as no timeout is triggered and the test goal is indeed unreachable). For Industrie 4.0, this is especially valuable, as test suites often have to conform to some criteria. For example, in the case of branch coverage as coverage criterion, this method allows creating a test suite reaching the branches if it is possible. For each branch that is not covered (if no timeout happened), the branch is unreachable and might be removed from the source code.

2.3.2 CPAchecker

For test-case generation, we utilize the software verification tool CPAchecker[3]. The main modules of CPAchecker used in this thesis are shown in Fig. 2.6 adapted from [21]. First, the source code of the system under test is parsed and translated into a CFA (see Sect. 2.3). Afterwards, different algorithms are used to perform reachability analysis.

[3] https://cpachecker.sosy-lab.org/

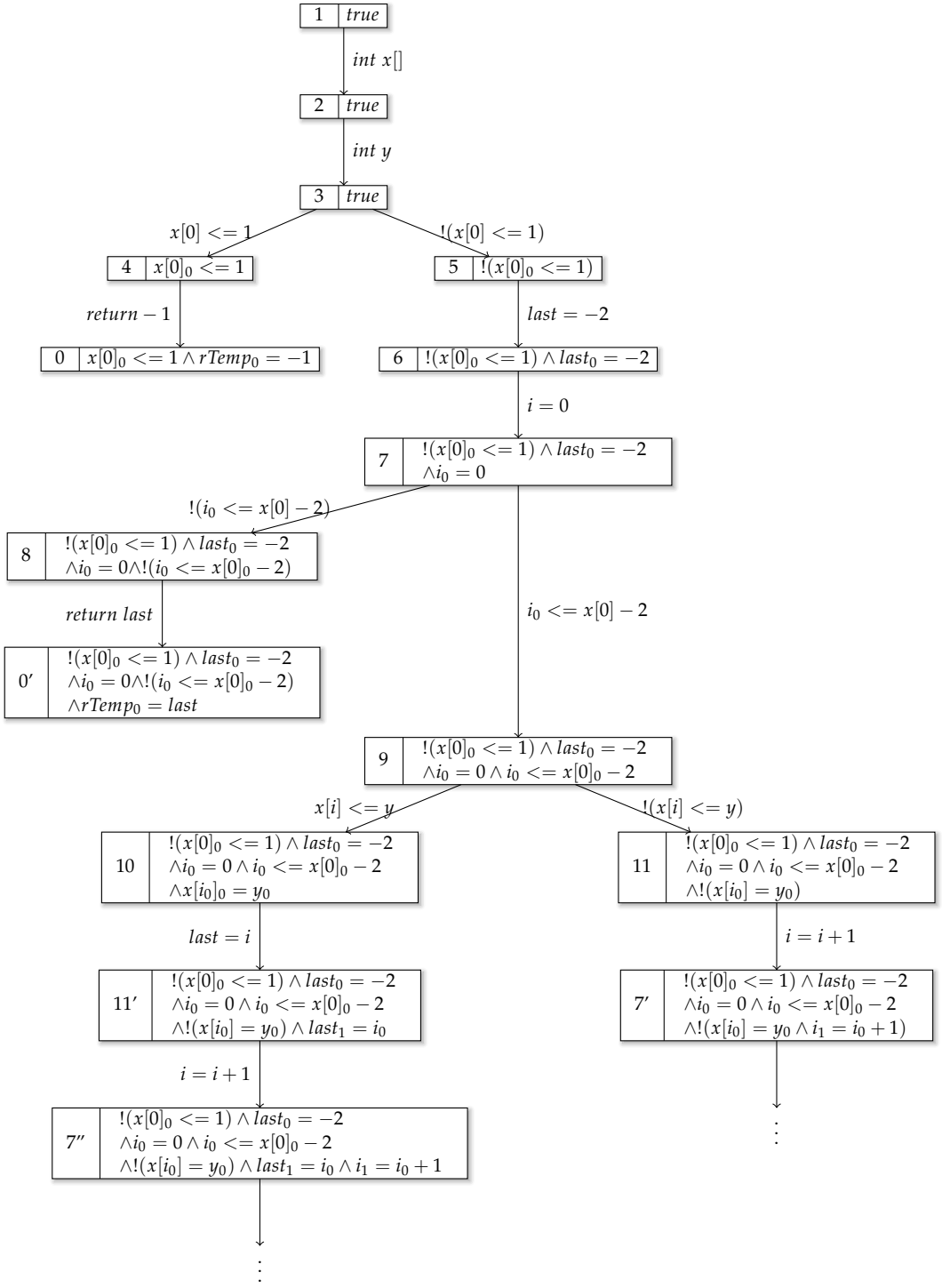


Figure 2.5: ARG of the C-function FIND_LAST

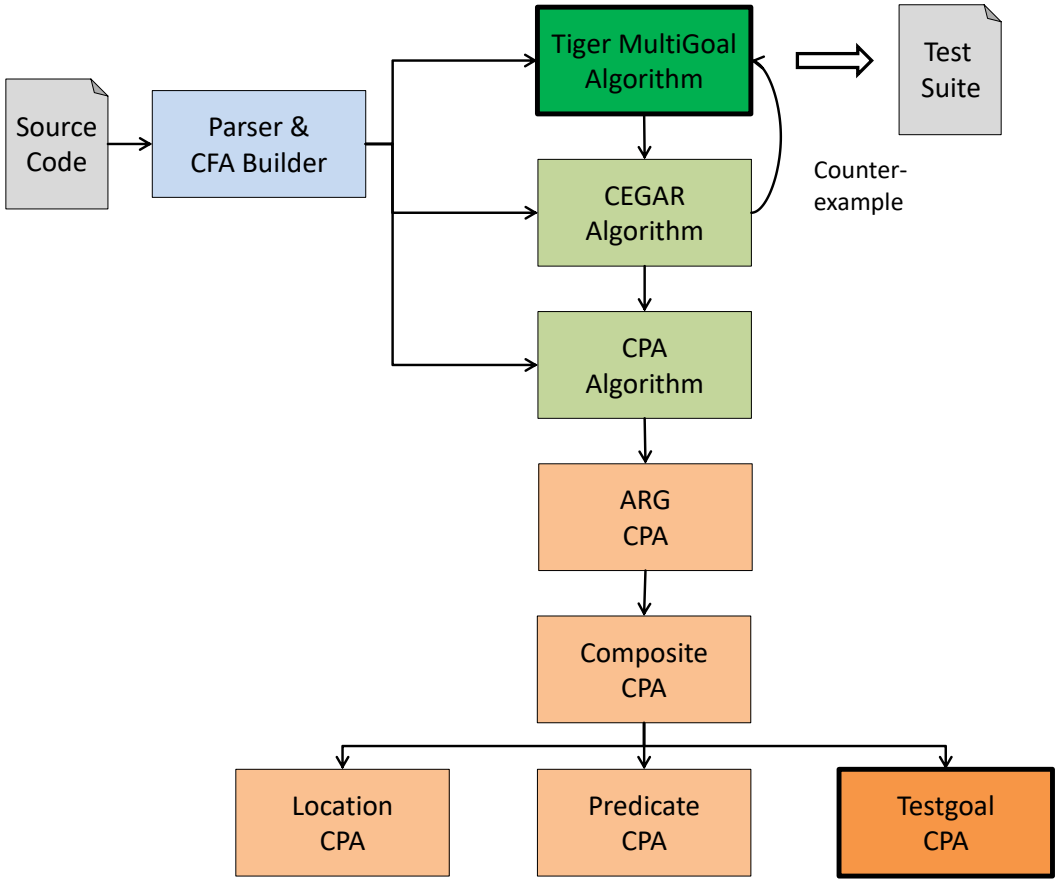


Figure 2.6: Modules for Test-Case Generation with CPAchecker (adapted from [21])

The reachability analysis then builds an abstract reachability graph, where each abstract state is a joint state built by different configurable program analyses (CPAs).

2.3.2.1 *CPAs*

Each CPA specifies an abstract domain of the program analyses and, therefore, a component of the abstract program states. In general, each CPA consists of the following components at least. [26]

- The **abstract domain**, which represents abstract program states.
- The **transfer relation**, which computes successor states of an abstract program state (given a corresponding CFA edge to the next program location).
- The **merge operator**, which specifies if and how two abstract states can be merged whenever the control flow merges.
- The **stop operator**, which specifies if an abstract state is covered by another abstract state and, therefore, does not have to be considered during reachability analysis anymore.

Additionally, a CPA can determine if an abstract state is considered as a violation of the specification that is considered during the reachability analysis (e.g., reaching a line of code that should not be reachable). This is done by marking the abstract state as *target state*. For this target state, a counterexample (i.e., a violation proof) is then created.

CPAchecker can be used, for test-case generation, by utilizing five different CPAs.

1. The ARGCPA, which is responsible for storing information about abstract states, their successors and predecessors, as well as computation of paths in the abstract reachability graph.
2. The CompositeCPA, which enables different CPAs to build joint abstract states and share information between abstract states of different CPAs.
3. The LocationCPA, which is needed to track the current program location during reachability analysis.
4. The PredicateCPA, which tracks and computes the constraints on variable assignments (and, therefore, also computes if an abstract state is reachable). This CPA is also used for computing a counterexample, which is explained later.
5. A TestGoalCPA, which tracks if an abstract state reaches a test goal and, therefore, marks the abstract state as target state.

In the following, we will further explain the components of the CPAs used throughout this thesis.

ARGCPA. The main idea of the ARGCPA is to store information about the abstract reachability graph and compute paths between abstract states. The ARGCPA always has another wrapped CPA (e.g., the CompositeCPA), which is responsible for building the abstract states. Therefore, the transfer relation, merge-operator, and stop-operator are mainly passed to the wrapped CPA. [13]

CompositeCPA. The CompositeCPA wraps multiple inner CPAs. This enables the wrapped CPAs to build joint abstract states and to share information. [26] For sharing information, the wrapped CPAs can implement a new component called the strengthening operator [35].

The abstract domain of the CompositeCPA is the joint abstract state of all wrapped CPAs. The transfer relation first checks the program location of the current abstract state (which means that at least one wrapped CPA needs to be used, which provides an abstract state with information about the program location). For each leaving CFA edge of the current program location (i.e., the CFA node), the CompositeCPA calls the transfer relation of the wrapped CPAs (with the CFA edge as an additional parameter). Therefore, the number of successors is at least the number of leaving edges of the current program location. Additionally, the CompositeCPA builds the cross-product of all successors of the wrapped CPAs per edge (which is usually only one successor). [35]

After the transfer relations of the wrapped CPAs have been called and the successors have been computed, the strengthening operator of the wrapped CPAs is called with the successor state of the CompositeCPA (i.e., the joint abstract state of all wrapped CPAs) as parameter. This enables the CPAs to further modify the resulting successors (i.e., strengthen) based on the information of the other abstract states computed by the other CPAs. [35]

The merge operator first checks if all wrapped CPAs agree that merging is possible. In this case, the merged abstract state will be the joint abstract state of all merged abstract states of the wrapped CPAs. [35]

Lastly, the stop-operator of the CompositeCPA checks whether a single wrapped CPAs stop-operator returns *true*. In this case, the stop-operator of the CompositeCPA returns *true* as well. [35]

LocationCPA. The abstract domain of the LocationCPA always corresponds to the concrete program location of the CFA. Therefore, the LocationCPA does not use abstraction of any kind. [32]

The transfer relation of the LocationCPA computes the successor of an abstract state correspondingly to the successors of the program location. Therefore, for each leaving edge of the successor, a successor is computed. The merge operator allows merging of all states with the same program location, and the stop operator checks whether the current program location has been reached once before. [32]

Therefore, if we start a reachability analysis with only the LocationCPA as wrapped CPA inside the ARGCPA (or the CompositeCPA), the resulting abstract reachability graph will be identical to the control flow automaton, as the abstract states of the LocationCPA correspond to nodes of the control flow automaton, and the ARGCPA uses the CFA edges as edges between abstract states. [32]

PredicateCPA. The abstract domain of the PredicateCPA corresponds to constraints over variable assignments (i.e., a predicate). The transfer relation computes the successor state based on the conjunction of the constraints of the current abstract state and the constraints (i.e., assignment or condition) of the CFA edge. The merge operator merges abstract states by joining the constraints via disjunction. [32]

The stop operator checks whether other abstract states already cover the current constraints (e.g., if two abstract states have the constraints $(a < 5)$ and $(a < 7)$, the constraint $(a < 7)$ would be covered already and does not have to be explored any further). [32]

Example 2.8. The abstract domain corresponds to the depiction of an exemplary ARG in Fig. 2.5. However, in this example, the ARG was not merged. As stated before, when merging, the constraints are joined via disjunction. For example, the abstract states 11 and 11′ in Figure. 2.5 could be merged as shown in Fig. 2.7. In this case, the equal part of the constraints (i.e., $!(x[0]_0 <= 1) \wedge last_0 = -2 \wedge i_0 = 0 \wedge i <= x[0]_0 - 2 \wedge !(x[i_0] = y_0)$) can be kept and the differencing parts (i.e., *true* and $last_1 = i_0$) are joined via disjunction. This enables a significant reduction of the ARG size and reduces (usually) the

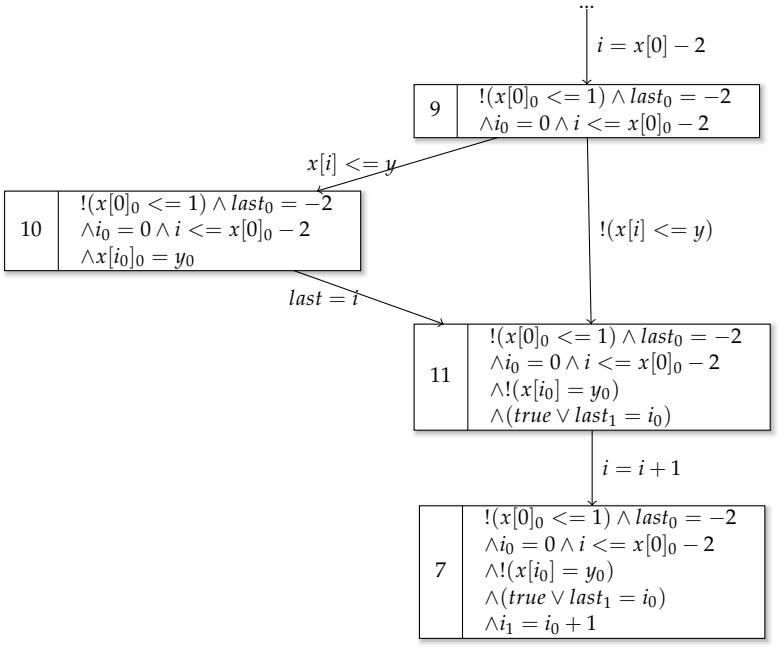


Figure 2.7: ARG of the C-Function FIND_LAST with CPAchecker

computation time as well, as other abstract states do not have to be computed multiple times (e.g., abstract state $7'$ from Figure. 2.5 is already covered by the merged state 7 in Fig. 2.7) .

Counterexample. If a CPA constructs an abstract state that is a target state, the CEGAR algorithm tries to compute a counterexample. In CPAchecker, the counterexample corresponds to a path within the ARG. The counterexample path is additionally annotated with explicit values for variable assignments which fulfill the path constraints. These values can be used to construct a test case from the counterexample. [30]

Example 2.9. Fig. 2.8 shows a counterexample for the abstract state at the program location 9 in Fig 2.5. The value for the assignment of variable x is $[2, 1]$, and the value for the assignment of variable y is 0. This counterexample would be computed for a test goal corresponding to the **`true-`**condition of the for-loop in our running example (see Fig. 2.3).

Additionally, a counterexample can be used to cover additional test goals. Each abstract state (and, therefore, each program location) in the counterexample is covered by the variable assignments. Therefore, each test goal that is within the path of the counterexample is also covered. In the case of the previous example, a test case corresponding to the $false$-branch of the first if-statements would also be covered. Thus, these test goals can be removed from the uncovered test goal set and ignored during the remainder of the reachability analyses. [40]

TestGoalCPA The TestGoalCPA is responsible for marking a successor as *targetstate* in case a test goal is reached. Therefore, the merge operator always

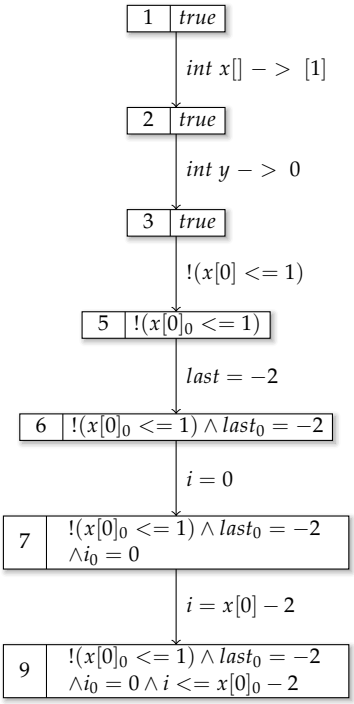


Figure 2.8: Counterexample for State 8 of the ARG in Fig. 2.5

allows merging and the stop operator never stops (i.e., always returns $false$). The transfer relation always returns the same successor as long as no test goal has been reached, as there is no information that needs to be stored. Otherwise, a *target state* is returned as successor.

2.3.2.2 *Algorithms*

Algorithms are key components that manage the reachability analysis.

CEGAR Algorithm. The CEGAR algorithm is used to refine reachability analyses. This enables us to start a reachability analysis with low precision (e.g., not storing information about variable conditions). In case a target state is reached within the reachability analysis, a counterexample is computed. If the counterexample is spurious (i.e., the counterexample exists only on the abstract model due to the lower precision, however, not on the program itself), a refinement is executed, and the reachability analysis is restarted (with increased precision). Otherwise, the counterexample is returned. This method is repeated until either a counterexample is found which is not spurious, or it can be proven that no specification violation (e.g., a test goal) is reachable. This method greatly increases efficiency (in terms of CPU time) as usually lower precision is still able to find a non-spurious counterexample and the reachability analysis with lower precision is much faster. [50]

CPA Algorithm. The CPA algorithm controls the construction of the abstract states. For this purpose, the CPA algorithm stores a set of reached states (i.e., abstract states which have been computed already) . Additionally, the reached

set has so-called waiting states, which are abstract states for which no successor has been computed so far. As long as there are waiting states, the CPA algorithm takes a single waiting state and lets the CPAs compute successors for this state. The successors are then included in the reached set and marked as waiting states.

Additionally, the CPA algorithm executes the merge, and the stop operator for the newly created successors and each state in the reached set with the same *partitionkey* as the successor. A partition key is built from each CPA which implements the Partitionable interface. In case of the LocationCPA, the key is the program location. In case of the PredicateCPA, the key is the abstraction. Therefore, the merge and stop operator are only executed for states with the same location and the same abstraction (if the LocationCPA and PredicateCPA have been enabled during reachability analysis). This prevents merging of states with different locations (e.g., merging state 7 and state 8 in Fig. 2.5). [26]

Tiger Algorithm. The Tiger algorithm is responsible for extracting the test goals from the CFA based on a coverage criterion and starting the reachability analysis for each test goal. If a counterexample is found for the test goals, a test case is extracted and written to the disk. Additionally, the Tiger algorithm is responsible for checking if the counterexample (and, therefore, the extracted test case) also covers other test goals. In this case, the newly covered test goals are ignored for all further reachability analyses. [33]

2.4 Regression Testing

The methods explained in the previous sections focused only on a single version of a system. However, during development (especially during agile development), program versions are frequently updated, often with small changes. Each time the program version is updated, it needs to be validated (i.e., tested) again. This process is called regression testing. [181] Especially, in Industrie 4.0 backwards compatibility is crucial and, therefore, needs to be validated as well. However, it is often infeasible to run the whole test-suite each time a change has been made since test-suites for large systems tend to run for a large amount of time. Especially, in Industrie 4.0 Scenarios tests are often executed on a physical machine. This additionally leads to material wear and increased maintenance costs. Therefore, only a fraction of the full test-suite can be executed. [181] To this end, different techniques have been subject to research to optimize testing for smaller changes. The main techniques for regression testing are the following. [181]

- Test-suite minimization: The removal of redundant test cases from existing test suites based on a new program version.
- Test-case selection: The selection of a subset of test cases from an existing test-suite tailored for testing a new program version.
- Test-case prioritization: Selecting an order for the execution of test cases to increase the probability of finding bugs faster.

```
6--: for (int i=0; i <= x[0]-2; i++)
6++: for (int i=1; i <= x[0]-2; i++)
```

Figure 2.9: First Bug Fix [154]

```
1  int find_last (int x[], int y) {
2    if(x[0] <= 1)
3      return -1;
4
5    int last = -2;
6    for (int i=1; i <= x[0]-2; i++)
7      if (x[i] <= y)
8        last = i;
9    return last;
10 }
```

Figure 2.10: New Program Version P_1 After Applying the First Bug Fix [154]

In the following chapters, we will explain these techniques in more detail. Furthermore, we will show a technique to create test cases based on program changes to further increase the effectiveness in terms of bug-finding potential of test cases.

2.4.1 *Running Example*

Until now, our running example consisted of exactly one program version. As mentioned before, the techniques explained in this chapter are based on at least two program versions. Therefore, we will now introduce the first bugfix of our running example.

Figure 2.9 shows the changes needed to fix the first bug i.e., to replace **`for (int i=0;i <= x[0]-2; i++)`** in line 6 with **`for (int i=1; i <= x[0]-2; i++)`**). Figure 2.10 shows the resulting program P_1.

2.4.2 *Test-Suite Minimization*

The focus of test-suite minimization is the removal of test cases from an existing test suite. Test cases that are not executable due to changes of interfaces (e.g., additional or removed parameters from function calls) are usually not part of test-suite minimization and need to be fixed or removed in a previous step. Usually, only test cases that became redundant (i.e., test cases that do not provide additional coverage to a specific criteria), due to changes to the source code, are removed. These test cases may become redundant, for example, due to the removal of code or changes in condition statement (and, therefore, changes in the paths). The test case would not test the intended path of the program but some different path, already tested by another test case. [181]

> **Example 2.10.** Consider a test suite consisting of the following test cases for our running example in version 0 (P_0).

1. Test case t_1: $t = \{([1], 0), -\}$
2. Test case t_2: $t = \{([4, 6, 6, 4], 5), -\}$

This test suite manages to reach 100% of branch coverage for P_0. However, after fixing the first bug (which results in version P_1 2.10), test case t_2 will not reach line 8 anymore. Therefore, the test suite will not provide 100% branch coverage anymore. To fix this issue, we can create a new test case $t = \{([4, 0, 1, 0], 0), -\}$. The new test case makes test case t_2 fully redundant, as t_1 and the new test case manage 100% branch coverage. Therefore, we can remove test case t_2 from the test suite without loss of coverage (concerning branch coverage).

As finding the minimal set of test cases without reducing the test-suite effectiveness (e.g., without reducing the coverage) is an NP-hard problem, there exist several heuristics to tackle this issue [88, 181]. However, even with these heuristics, test-suite minimization can be quite time-consuming. Therefore, it is often used after several program modifications and not after each one. As program modifications are often small changes, test-suite minimization would not be able to remove many test cases after each modification anyway. However, executing test-suite minimization after multiple program modifications still helps to keep the test suite at a manageable amount of test cases. After test-suite minimization, the test-suite should be checked if additional test cases are needed (e.g., if the coverage criteria is not fulfilled due to the source code changes and, therefore, new test cases are needed). [181]

2.4.3 *Test-Case Selection*

In contrast to test-suite minimization, test-case selection does not alter the original test suite. The focus is a temporary test suite (a subset of the original test suite) tailored for the changes made by the current program modification (e.g., for our running example the test suite would be tailored to test the changes made to line 6). [181]

Test cases that do not reach the modified parts of the code are irrelevant for testing the new program version. They cannot show any different behavior (see RIPR criteria 2.2.4) and cannot detect newly introduced bugs. For this reason, executing the whole test suite for each program modification is inefficient, and test-case selection helps in increasing the efficiency of this testing step.

Example 2.11. Consider a test suite consisting of the following test cases for our running example.

1. Test case t_1: $t = \{([1], 0), -\}$
2. Test case t_2: $t = \{([4, 6, 6, 4], 5), -\}$

The first bug fix in our running example only changes line 6. As the first test case (t_1) will never reach the changes, it cannot reveal any differences

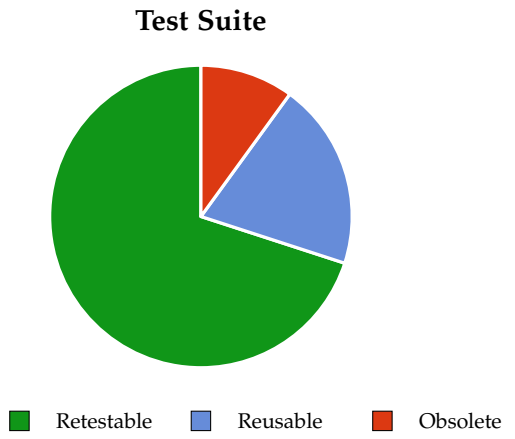


Figure 2.11: Test-Suite Categorization (adapted from [118])

> and, therefore, cannot find any newly introduced bugs (e.g., introduced by a faulty bug fix). In this case, we only have to execute test case t_2.

2.4.4 *Test-Case Prioritization*

The focus of test-case prioritization is the order of execution of test cases. The goal is to find bugs as fast as possible and, therefore, execute test cases with a higher bug finding potential first. Additionally, the execution time of each test case can also be taken into account to optimize the prioritization further. In this case, the goal is to find an optimal solution for two objectives (i.e., multi-objective optimization). [181] However, annotating test cases with a value for bug-finding potential is usually not easy. First, the information of locations of potential bugs is needed. To this end, mutation testing can be used to simulate bugs and check which test case is able to detect the mutation. Additionally, the execution time of the test cases can be measured as well, which enables test-case prioritization to chose test cases based on their execution time and bug-finding potential.

Test-suite minimization, selection, and prioritization can be combined during regression testing. Test suites can consist of reusable (i.e., test cases not needed for the current program modification but maybe later), retestable (i.e., test cases needed for the current program modification), and obsolete/redundant (i.e., test cases that have become redundant due to code changes) test cases (see fig. 2.11 based on [118]). Therefore, test-suite minimization can be used to remove obsolete test cases. Test-case selection can then be used to select retestable test cases and test-case prioritization can be used to prioritize retestable test cases. As each step reduces the number of test cases, the following steps are easier to perform (in terms of efficiency).

In industrial applications (e.g., Industrie 4.0) test case execution is often applied on a physical machine. Therefore, test execution is often accompanied with human working time, energy consumption (of the machine, which is often higher than energy consumption of PCs) and physical wear and tear. For this reason,

```
void find_last_p0_1(int x[], int y){
  if(find_last_p0(x,y) != find_last_p1(x,y))
    test_goal:printf("differencing_test_case_found");
}
```

Figure 2.12: C-Function to compare Behavior of Running Example Versions

test-suite minimization, selection and prioritization is especially important for Industry 4.0.

However, as test-suite minimization should be the first step for reducing testing effort, the main focus of this work is test-suite minimization, while test case selection and prioritization are out of scope.

2.4.5 *Regression-Test-Case Generation*

Test-case generation in chapter 2.3 only takes a single program version into account. Test goals have been the reachability of single program locations (e.g., statements or branches). As mentioned before, the detect a bug, reaching the bug is not sufficient (see RIPR criteria 2.2.4). The bug also needs to change the program state and must be observable (i.e., change the observable behavior).

This specification can also be used for test-case generation. To this end, the old and the new program version can be merged into a single program to generate a test case that ensures different behavior when executed on the old version as compared to executing on the new version. As only these revealing test cases can detect bugs, the chances of detecting bugs are in general higher for test cases created in this manner. This is also called *differential test-case generation*. [132] However, the change might also be intentional (e.g., bug-fix, new features, etc.). Additionally, the test case often needs to be executed as well. For example, in Industrie 4.0 backward compatibility is often critical. However, a change in the behavior does not always lead to a change in the whole system (e.g., for the laser-plotter in our example, the behavior of functions might change, however, as long as the resulting plot remains the same for the same input, backward compatibility is retained).

Example 2.12. In the case of our running example, both version (P_0 2.3) and P_1 2.10) can be merged, and a new function for comparison purposes can be added. This new function is depicted in fig. 2.12 and takes the same arguments as the running example's function. It then executes both versions of the running examples and compares the result. If the result differs, the function prints the string "$differencing\ test\ case\ found$" to the console. By generating a test case with the print-line as test goal, we will create a test case that will result in a different output value for both versions (with the same input values).

In case of more complex parameters, the values might need to be copied and passed to the functions to prevent the overriding of values. Additionally, the state of global parameters can be used to check for different behavior of

> the functions (and also might need to be reset before executing the function of the newer version).

2.5 Software-Product Lines

In industry, and especially Industrie 4.0, there exist multiple machines with similar functionality (e.g., different types of plotters, as supported by our running example, see Sect. 2.1.3).

The software to operate these different machines (i.e.,their drivers) often share common code parts (e.g., the axis-control in our running example). Often these parts of the code are copied from existing software, however, this makes maintenance difficult. For example, if a bug is found in the copied parts of the code, the bug needs to be fixed in multiple projects.

To facilitate re-use and reduce development and maintenance costs the software of similar systems can be merged (or implemented as) to a software-product line (SPL). A software-product line consists of multiple components (so-called features), which can be combined in different ways.[14] Each configuration, specifying the combination of these features, leads to a different resulting product (e.g., specifying the plotter in our running example with a laser, without a pen and with a paper feed). The software-product line comprises all product, which can be build from a valid selection of features. Therefore, the SPL allows the simultaneous development of multiple products, with common code parts. This reduces development effort and especially maintenance costs. [51]

Running Example. To further elaborate different aspects of SPLs, we will enhance our running example with new functionality, implemented as SPL. Figure 2.13 shows the new running example. To denote different features, we use preprocessor directive (e.g., if feature $ISEMPTY$ is selected the resulting product includes the lines between 13 and 16).

The running example incorporates the feature $ISEMPTY$, which leads to a product checking whether the array is empty (and returning the error code -1) or not. Selecting the feature $FIRST$ leads to a product which searches for the first occurrence of y in x, selecting feature $LAST$ will search for the last occurrence. This is supported by reversing the iteration order, depending on the selected feature.

Lastly, feature $COUNT$ will instead count the occurrences of y instead of returning the index of the first or last occurrence.

To fully use the potential of software-product lines, the re-use of implementation artifacts (e.g., source code) should be maximized. To this end, software-product-line engineering (SPLE) was developed[51]. Next, we will further elaborate the idea of SPLE.

2.5.1 *Software-Product-Line Engineering*

The goal of the developlment process of software-product lines the the development of a group of similar product as one. Therefore, the development process of

```
1  int c;
2  int startIndex;
3  int lastIndex;
4  int increment;
5  int last;
6
7  int find_configurable (int x[], int y) {
8    #if defined(COUNT)
9    c = 0;
10   #else
11   last = -2;
12   #endif
13   #if defined(ISEMPTY)
14   if(x[0] <= 1)
15     return -1;
16   #endif
17   #if defined(FIRST) || defined(COUNT)
18   startIndex = x[0]-2;
19   lastIndex = -1;
20   increment = -1;
21   #elif defined(LAST)
22   startIndex = 0;
23   lastIndex = x[0]-1;
24   increment = 1;
25   #endif
26   for (int i = startIndex; i != lastIndex ; i += increment){
27     if (x[i] <= y)
28       #if defined(COUNT)
29       c++;
30       #else
31       last = i;
32       #endif
33   }
34   #if defined(COUNT)
35   return c;
36   #else
37   return last;
38   #endif
39 }
```

Figure 2.13: Initial SPL Version $P_{0_{SPL}}$ (adapted from [155, 154])

```
1  int c;
2  int startIndex;
3  int lastIndex;
4  int increment;
5  int last;
6
7  int find_configurable (int x[], int y) {
8    int last = -2;
9    if(x[0] <= 1)
10     return -1;
11
12   int startIndex = x[0]-2;
13   int lastIndex = -1;
14   int increment = -1;
15
16   for (int i = startIndex; i != lastIndex ; i += increment){
17     if (x[i] <= y)
18       last = i;
19   }
20   return last;
21 }
```

Figure 2.14: Derived Product of the SPL Running Example

single products is not adequate anymore, as the requirements change. SPLE consists of two major development processes, domain engineering and application engineering. [144]

Domain Engineering. Domain engineering is responsible for designing (reusable) artifacts necessary for building the SPL. To this end, domain engineering is concerned with the design of all products, and, therefore, which features are needed. Additionally, domain engineering is responsible for the implementation of the features. However, domain engineering is not concerned with the implementation of resulting products, only with the implementation of the features. [14, 144]

Application Engineering. The task of the application engineering is to build specific products fulfilling requirements of customers. To this end, features developed in domain engineering are selected and a product consisting of those features is derived. Deriving products from a SPL with a configuration for the features can also be automated (e.g., using a preprocessor in case of our running example in Fig. 2.13). [14] An example of a derived product for our running example is shown in Fig. 2.14, where the features *ISEMPTY* and *FIRST* are selected. Lastly, application engineering is concerned with validation and verification of derived products.

2.5.2 *Feature Models*

Obviously, not every single combination of features will lead to a useful, or even usable, product. For example, selecting feature *FIRST* and feature *LAST* in the

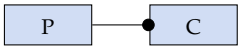
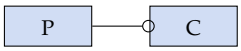
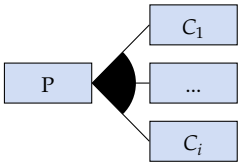
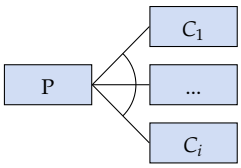
Relationship	Representation	Propositional Logic
Mandatory		$P \iff C$
Optional		$C \implies P$
Or		$P \iff (\vee_{x=1}^{i} C_x)$
Alternative		$(P \iff \vee_{x=1}^{i} C_x) \wedge$ $(\wedge_{1 \leq x < j \leq i} \neg C_x \vee \neg C_j)$

Table 2.1: Feature-model Components (adapted from [173])

running example in Fig. 2.13 will lead to the same program as selecting feature *FIRST* without feature *LAST*, rendering the selection of *LAST* useless.

The supported interactions between features can be specified using different models[54]. One of the most prominent ways to specify the interaction is by using so-called feature models[14]. To design feature models in a graphical fashion, feature diagrams can be used [107].

Table 2.1 (adapted from [173]) shows the most commonly used relationships between features, as well as their translation to propositional logic, which will be needed later. First, a mandatory relationship between feature P and feature C states that if any of those features is selected, the other needs to be selected as well. Second, an optional relationship between feature P and feature C states that if feature C is selected, P needs to be selected as well, however not vice versa.

An or-group describes a group of features from which one or more needs to be selected, in case the parent feature is selected as well (e.g., if feature P is selected at least one of the features C_1 to C_i needs to be selected as well). Additionally, if any feature from that group is selected, the parent feature needs to be selected as well.

Lastly, an alternative group describes a group of features from which exactly one feature needs to be selected in case the parent feature is selected as well. Additionally, if a feature from that group is selected, the parent feature needs to be selected as well.

To specify constraints on feature without parent-child relationships, so-called cross-tree-constraints can be used. Cross-tree constraints allow the definition of arbitrary relationships between features expressed by propositional logic.

An exemplary feature model for our running example is shown in Fig. 2.15. In this example, the features *FIND* and *SEARCH* are marked as abstract features, as they do not represent actually implemented features. They are used solely for

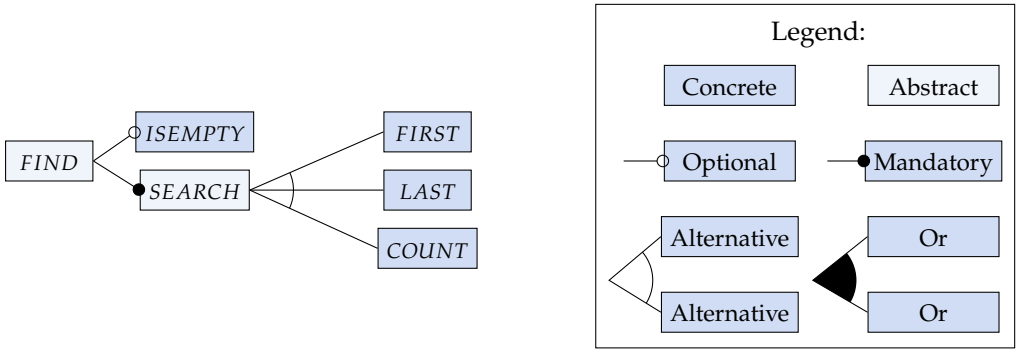


Figure 2.15: Feature Model of the Running Example in Fig. 2.13 (based on [155])

Product/Config.	FIND	ISEMPTY	SEARCH	FIRST	LAST	COUNT
P_1/C_1	x	-	x	x	-	-
P_2/C_2	x	x	x	x	-	-
P_3/C_3	x	-	x	-	x	-
P_4/C_4	x	x	x	-	x	-
P_5/C_5	x	-	x	-	-	x
P_6/C_6	x	x	x	-	-	x

Table 2.2: Products/Configurations of the Running Example shown in Fig. 2.13

modeling purposes of the feature model. Feature *SEARCH* is mandatory, and, therefore, needs to be selected for each product. Feature *ISEMPTY* is optional, meaning it can be selected or deselected independent of other features. Lastly, the features *FIRST*, *LAST* and *COUNT* are grouped as alternatives and, therefore, exactly one of them has to be selected. This leads to a total of six products for our running example, shown in Tab. 2.2.

2.5.3 *Software-Product-Line Testing*

As software-product lines are enriched with variability, standard software testing becomes insufficient. Due to the interaction between features, some bugs might only occur in specific products. Therefore, to fully test the SPL the whole family of products needs to be tested. However, the number of products usually grows exponentially with the number of features, testing each product becomes infeasible. To this end, multiple alternatives have been subject to research in the past. The most established techniques are elaborated in the following. [123]

Product-wise Testing The idea of product-wise testing is to use conventional (non-SPL) testing technique for each product. Testing each product leads to a fully validated SPL, as there are no valid feature interactions that remain untested. However, as mentioned before, the number of valid product usually grows exponential with the number of features. As there is no re-use of information about the tests of different products, the testing efforts often also grows ex-

```
1 void isvalid () {
2   if(FIND &&
3   (ISEMPTY || !ISEMPTY)
4   && ((FIRST && !LAST && !COUNT) ||(!FIRST && LAST && !COUNT) || (!FIRST && !LAST
        && COUNT))  ){
5   return;
6   }
7   exit(0);
8 }
```

Figure 2.16: IsValid Function for the Running Example

ponentially. Therefore, this testing methodology is often infeasible for real-world projects.

As shown in Tab 2.2, our feature model allows 6 valid configurations and, therefore, the derivation of 6 different valid products. Therefore, the whole testing process for our running example would need to be executed six times.

Sample-based Testing Sample-based testing tries to tackle the issues of product-wise testing by reducing the number of products for validation. The idea is to use a representative set of products, from which the results of the testing process can be deduced for the remainder of the products. However, the main challenge is the selection a representative set of products. Additionally, there are no guarantees about bugs in untested products. For example, bugs only appearing for certain feature combination can only be found when testing product with those feature combinations. However, sample-based testing allows for a control in effectiveness and efficiency of testing by selecting varying numbers of products for validation. The most effective set of products would result in product-wise testing, which leads to the most testing effort. The least testing effort for sample-based testing would be to test only a single product, which would lead to the least effective testing, as very few feature interactions would have been tested. To this end, building a representative set of products has often been subject to research. [62]

Incremental Testing The idea of incremental testing is to test differences between products. To this end, first a single product is fully tested. Next, the differences for the remaining products are tested. Similar to regression testing, the goal is to reduce testing effort by only testing the delta of two products. [124] This methodology greatly decreases testing effort for each product, however, as the number of products grows exponential with the number of features, this technique still might be infeasible for projects with large amounts of features.

Family-based Testing Family-based testing is concerned with testing the whole SPL in a single step (i.e., testing all products at once). To this end, a model is used, which incorporates all information about the SPL. A model for this could also be plain C code, as it is possible to include the feature model into the implementation by translating the satisfying condition to C code. The propositional logic of the feature models can easily be translated, as
$a \iff b \equiv a \implies b \wedge b \implies a$

and $a \implies b$ can be translated to $\neg a \vee b$. Using these translations, we can compute a C function corresponding to a given feature model.

For example, for our running example we can create a function **`isValid`**, which evaluates whether the feature selection is valid (see Fig. 2.16).

For such a model family-based testing creates test-cases which incorporates the products (i.e., feature selections) for which they are valid. Those test-cases can often be executed on multiple products (e.g., a test case covering test goals in a core part of the code is executable for every product, for example covering line 11 in our running example). However, the downside of family-based testing is the testing itself (although only executed once) might be more expensive than testing all products in succession, as the model under test is more complex compared to single products. [40]

3

TEST-CASE GENERATION FOR SOFTWARE PRODUCTS

In this chapter we will introduce new techniques for automated test-case generation for software products. To this end, we will provide an overview of existing state-of-the-art techniques. Next, we will further optimize these techniques in multiple ways. The contribution of this chapter is the optimization of test-case generation in terms of efficiency (i.e., CPU time and test-suite size), as well as effectiveness (actually achieved coverage based on a coverage criterion). To this end, the following challenges will be tackled in this chapter.

- How to increase effectiveness of test-case generation in terms of actual coverage achieved based on a given coverage criterion (**C1**).
- How to increase efficiency of test-case generation...

  ...in terms of CPU time needed to generate test suites (**C2.1**).

  ...in terms of the number of test cases generated (**C2.2**).

As mentioned in the previous chapter, we will utilize the framework CPAchecker for test-case generation purposes. The techniques developed to optimize test-case generation have been implemented in CPAchecker for testing and evaluation of the techniques.

The contents of this chapter are based on the following publications:

> [156] Sebastian Ruland, Malte Lochau, Oliver Fehse, and Andy Schürr. CPA/Tiger-MGP: test-goal set partitioning for efficient multi-goal test-suite generation. In *International Journal on Software Tools for Technology Transfer*, Springer Science and Business Media (LLC), 2020. doi: 10.1007/s10009-020-00574-z.

> [157] Sebastian Ruland, Malte Lochau, and Marie-Christine Jakobs. HybridTiger: Hybrid Model Checking and Domination-based Partitioning for Efficient Multi-Goal Test-Suite Generation (Competition Contribution). In *Proceedings of the 21st International Conference on Fundamental Approaches to Software Engineering*, pages 520-524. Springer International Publishing, 2020. doi: 10.1007/978-3-030-45234-6_26.

Test Goals. For the remainder of this thesis, test goals are encoded as lists of CFA edges, if not stated otherwise. These test-goals are based on a given coverage criterion. For example, to create a test case to reach a certain line of code (e.g.,

line 3 in our running example 2.3), the test case is encoded as a list of CFA edges with a single element, namely the edge ((4) $\xrightarrow{return\ -1;}$ (0))

Currently, there are several control-flow based coverage criteria implemented (e.g., branchcoverage, statement coverage, etc.), as well as a syntax to define labels in code that should be used as test goals (e.g., to reach an erroneous location). More complex control-flow based coverage criteria, such as MC/DC, can still be achieved by utilizing a-priori program transformation techniques [78]. Data-flow based coverage criteria are currently out of scope.

3.1 Multi-Goal Checking

The default strategy to generate test cases with software model checking techniques is to check each test goal in isolation. For this test goal, a reachability analysis is executed, and if the test goal is reachable, a counterexample is computed and a test case extracted. However, if the coverage criterium leads to a huge number of test cases (e.g., branch coverage), the number of executed reachability analyses is huge as well. To increase the efficiency of test-case generation, previous studies have utilized techniques from multi-property checking to check multiple test goals in a single reachability analysis. As soon as the reachability analysis shows the reachability of a test goal, a counterexample and test case are generated. Next, the reachability analysis is resumed for the remainder of the test goals. [15]

Abstract State Merging There exist different coverage criteria and, therefore, different criteria for single test goals. While many coverage criteria lead to test goals that can be represented as a single CFA edge, there also exist criteria where a single CFA edge as test-goal representation is insufficient. For example, in our running example, a test goal could be to first reach line 8 to assign a value to *last* and afterwards reach the line 9 to return the value. This is represented as a list of CFA edges which have to be traversed in order to finally fulfill the test goal (e.g., [(10) $\xrightarrow{last = i}$ (11), (8) $\xrightarrow{return\ last}$ (0)]). However, simply running a reachability analysis as described in the previous section can lead to a loss of information and, therefore, to incorrect test cases.

Example 3.1. Consider the test goal

- tg_1=[(9) $\xrightarrow{x[i] <= y}$ (10), (11) $\xrightarrow{i = i + 1}$ (7)].

In this case, the test goal would be fulfilled in the abstract state 7 (see Fig. 2.7) if the *true*-branch of if-statement inside the *while*-loop has been taken before. However, the information which path was taken to reach abstract state 7 is not present anymore due to the merge. Therefore, a valid counter-example could take any of the previously explored paths.

To keep this information, it can be weaved in the abstract states during the reachability analysis (so-called on-the-fly weaving) [15]. As shown in Fig. 3.1, new abstract states and new CFA edges have been weaved in the ARG. These new

CFA edges include additional information for the test goal by incrementing the variable g_0 and enforcing the value of g_0 to be equal to 2 before reaching the final abstract state. Therefore, a counterexample computed for abstract state 7′ is forced to take the left path of the ARG and will cover the test goal. After creating a counterexample for this specific test goal, the last abstract state (7′) needs to be removed from the ARG, as otherwise all further successors would be forced to take the left path of the ARG.

The actual method to weave information in the ARG is later explained in section 3.3.

3.2 Efficient Test-Case Generation

This section is concerned with the optimization of efficiency in test-case generation in terms of CPU time. To this end, we will illustrate multiple methodologies, which have been developed during the course of this thesis. First, we will illustrate the technique called partitioning (i.e., grouping of test goals) and partitioning strategies. Additionally, we will present the implementation and show and discuss the evaluation of the different strategies. Next, we will further optimize the efficiency of test-case generation by utilizing hybrid model checking techniques [100] combined with partitioning. Hybrid model checking combines different analysis techniques to use the strengths of the different techniques to cover additional test goals. Lastly, we will present and discuss the implementation and evaluation of this technique as well.

3.2.1 *Partitioning.*

The obvious advantage of multi-property checking is to execute fewer reachability analyses for test-case generation. This is especially advantageous for larger amounts of test goals. However, there are different disadvantages as well. The main disadvantage is that the abstraction possibilities during the reachability analysis are lower, as more information needs to be tracked. This slows the reachability analysis down. This loss of efficiency increases even more if information about test goals needs to be weaved in the abstract reachability graph. To conclude, a single reachability analysis for each test goal is inefficient, as the number of reachability analyses is high. However, a single reachability analysis might also be inefficient, as the amount of information that needs to be tracked is high.

To this end, partitioning is concerned with grouping (i.e., partitioning) of test goals, which (hopefully) share a high amount of information that needs to be tracked. Therefore, these test goals can be tried to cover in a single reachability analysis. The number of reachability analyses that need to be executed is equal to the number of partitions created during partitioning. This constitutes a trade-off, as the number of reachability analyses decreases compared to a single analysis per test goal. However, the amount of information that needs to be tracked in a reachability analysis is less than for standard multi-property checking.

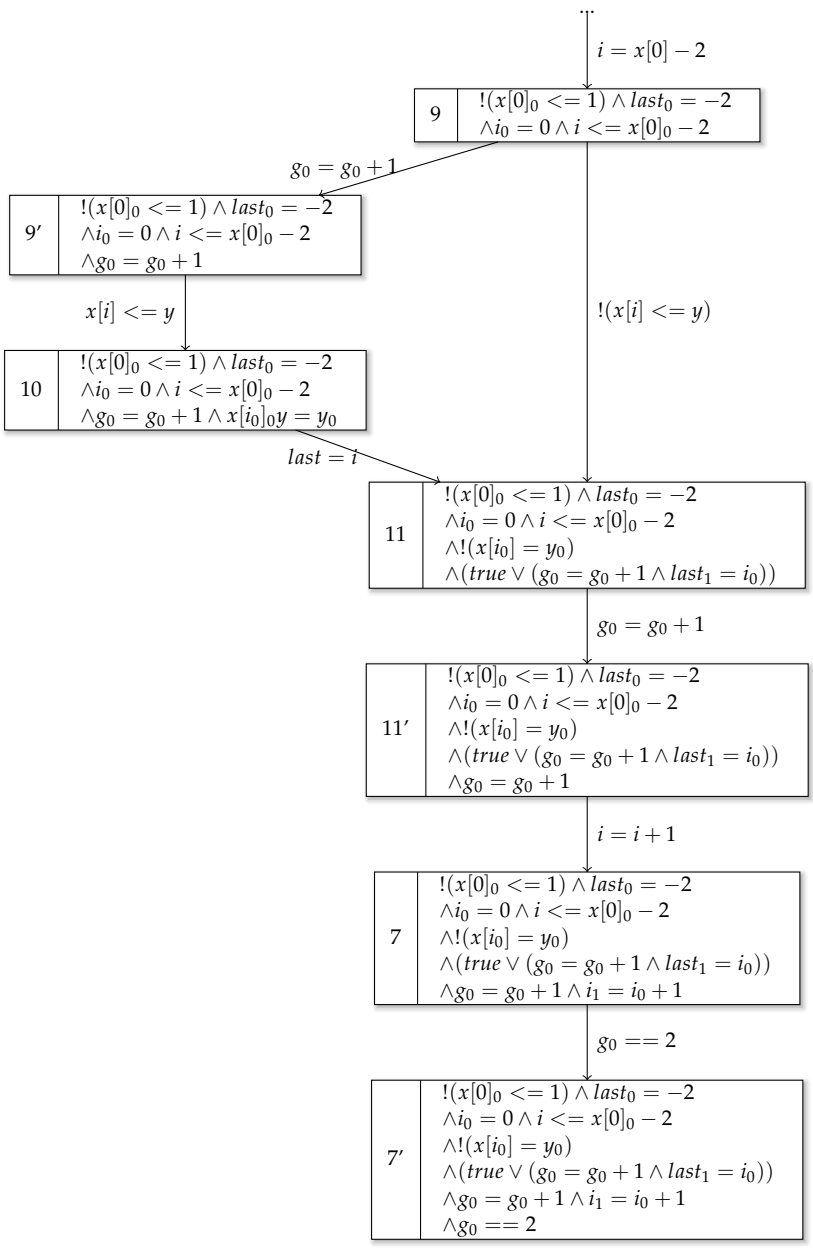


Figure 3.1: ARG of the C-Function find_last with CPAchecker

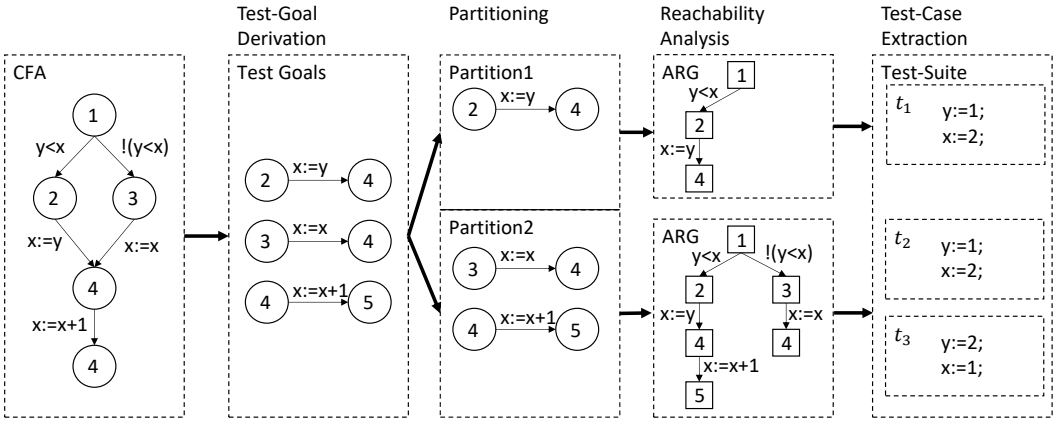


Figure 3.2: Partitioning Overview (adapted from [156])

3.2.1.1 *Overview*

Figure 3.2 (adapted from [156]) shows the overview of the new partitioning approach. First, the parser and CFA generator of CPAchecker is utilized to generate the CFA. Next, the Tiger Algorithm is used to derive the test goals from the CFA based on the coverage criterion. In case of the example in Fig. 3.2, statement coverage is used as coverage criterion. Next, the test goals are grouped into different partitions based on the selected partitioning strategy and partition size. Lastly, a multi-property reachability analysis is executed for each partition, and the test cases are merged in a shared test suite. Usually, the test-case generation process is accompanied by a timeout. Therefore, the reachability analysis for each partition is also limited in terms of CPU time (depending on the timeout strategy), usually based on the time limit for the test-case generation divided by the number of partitions.

To partition the test goals, different kinds of information can be used. In case of random partitioning, no information about the test goals is needed. However, the partitioning can also be performed based on local or global control-flow information or even data-flow information. While local and global control-flow information about the CFA are easy to obtain, data-flow information would require an a-priori data-flow analysis before partitioning. Since partitioning tries to provide an increase in efficiency, the partitioning itself should not take much CPU time (as this time is not available for the reachability analysis anymore). Therefore, data-flow analysis has been out of scope for this thesis.

Lastly, while partitioning enables parallelization by design (i.e., each reachability analysis for a single partition can be run on a separate CPU core), the main focus of this chapter is efficiency in terms of total CPU time.

3.2.1.2 *Partitioning Strategies*

In this section, we will further explain different partitioning strategies, which have been developed during this thesis. We will illustrate the concepts of the different strategies as well as their strengths and weaknesses. The partitioning strategies

are divided based on the information used. First, we will describe random-based partitioning, which uses no information at all. Next, we will show different partitioning strategies based on local information (i.e., information that is present in the test-goals themselves). Lastly, we will describe partitioning strategies based on global information about the CFA (e.g., information about CFA paths, etc.). Additionally, the size of the partitions will be either a total amount or a relative amount based on the overall number of test goals.

Example 3.2. To further illustrate the strategies and concepts of the remainder of this section, we will now define a set of test goals for our running example (see Fig. 2.3).

- tg_1: (3) $\xrightarrow{x[0] <= 1}$ (4)
- tg_2: (3) $\xrightarrow{!(x[0] <= 1)}$ (5)
- tg_3: (7) $\xrightarrow{!(i <= x[0] - 2)}$ (8)
- tg_4: (7) $\xrightarrow{i <= x[0] - 2}$ (9)
- tg_5: (9) $\xrightarrow{x[i] <= y}$ (10)
- tg_6: (9) $\xrightarrow{!(x[i] <= y)}$ (11)

These test goals will be used to show the different behavior of the partitioning strategies and partition sizes.

3.2.1.3 *Random Partitioning*

The simplest partitioning is random-based. Random partitioning randomly selects test goals and adds them to the current partition. This step is repeated until the maximum number of test goals per partition is reached and a new partition is created. This is done until all test goals have been added to a partition. As the partitioning is random, the efficiency of test case generation may vary between different executions.

Example 3.3. If the size of the partitions is 3 for our example, the test goals would be grouped into 2 different partitions randomly. A possible result of the partitioning would be $\{tg_2, tg_4, tg_5\}\{tg_1, tg_3, tg_6\}$.

The obvious advantage of this strategy is the simplicity and efficiency as random selection is very fast. Therefore, the efficiency of the partitioning itself is high and consumes nearly no CPU time. In case of a time limit, the CPU time can be nearly fully used for the reachability analyses. Additionally, random is always a decent baseline when comparing different methods. If another strategy is less efficient compared to random, this suggests either design or implementation flaws within the strategy.

3.2.1.4 *Local Information-based Partitioning*

Local information-based partitioning only takes the information into account that is directly available based on the test goals themselves. This means that no extra information about the CFA, Paths, etc., is considered. The clear strength of this approach is the efficiency of the partitioning itself. As the information is available without further computation, the only computational effort is the partitioning algorithm itself. However, the disadvantage is that available information (even though it would need to be computed first) is not considered (e.g., CFA path information about the test goals). This might lead to less efficient reachability analysis when compared to global information-based partitioning.

Domination-based Partitioning. The first local information-based partitioning strategy is called domination-based partitioning. This strategy partitions the test goals based on the post-order ID of their successor CFA node. Figure 3.3 shows the CFA of our running example with the corresponding post-order IDs of the nodes (the values below the lines in the CFA nodes). In this example, the CFA node with the ID 1 has the post-order ID 11. The post-order ID is a unique ID, which also provides one guarantee. A node with a smaller post-order ID is never a predecessor of a node with a larger post-order ID in any path possible within the CFA. Therefore, the chances are high that a CFA node with a smaller post-order ID is deep in the CFA.

The domination-based partitioning uses this guarantee to try to group test goals in a single partition which are deeper in the CFA. This means that if a counterexample is found, the chances are high that there are multiple test goals within the counterexample path and can also be removed from the set of uncovered test goals (see Sect. 2.3.2.1).

> **Example 3.4.** We will again split the test goals into partitions with three test goals per partition for this example. For domination-based partitioning, we first select the test goal with the lowest post-order ID, which is test goal g_6. Next, we add this test goal to the first partition and remove it from the set of test goals for partitioning. This step is repeated until the first partition is filled. Afterwards the next partition is created, and the previous steps are repeated. This leads us to the partition result with the first partition containing the test goals {tg_6,tg_5, tg_4} and the second partition containing the test goals {tg_3, tg_2, tg_1}.

Reverse Domination-based Partitioning. The possible disadvantage of the domination-based partitioning is that the first partition contains test goals that are probably difficult to cover (i.e., take a lot of CPU time). In this case, the timeout of the partition might be reached before covering a lot of test goals. In contrast, if a partition with easy-to-cover test goals is started first and the partition takes less time, the remainder of the CPU time can be used for the more complex test goals.

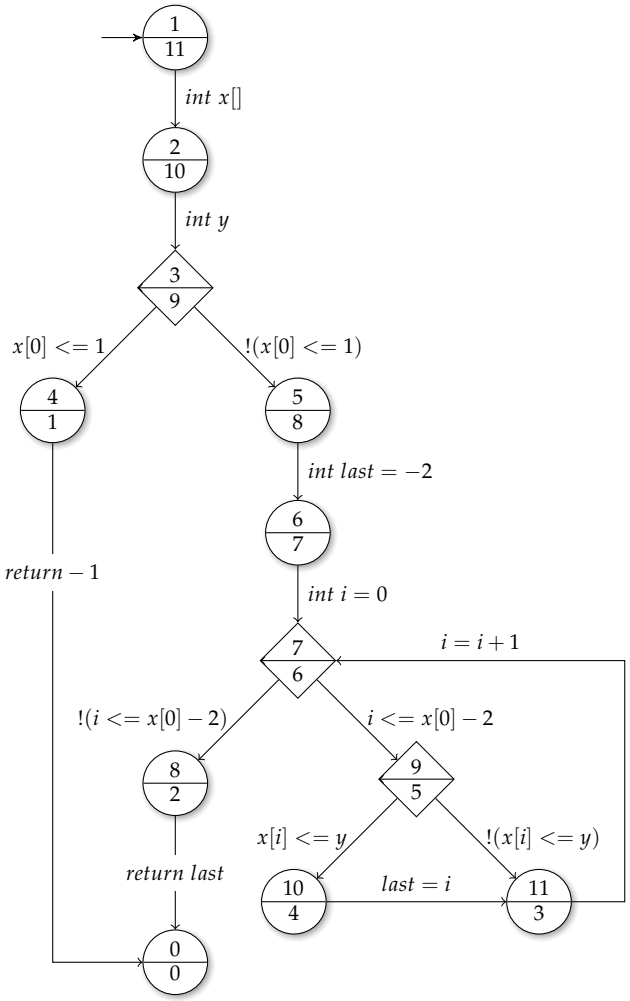


Figure 3.3: CFA of the C-Function FIND_LAST with Post-Order Ids

This is the motivation for the reverse domination-based partitioning strategy. As stated before, the test goals with lower post-order ID are likely harder to cover. Therefore, reverse domination-based partitioning also takes the post-order ID into account, however, in reversed order.

> **Example 3.5.** Again, we will use partitions with the size of three test goals per partition. For reverse domination-based partitioning, we first select the test goal with the highest post-order ID, which is test goal g_1. Again, we add this test goal to the first partition and remove it from the set of test goals for partitioning. This step is repeated until the first partition is filled. Afterwards the next partition is created, and the previous steps are repeated. This leads us to the partition result with the first partition containing the test goals {tg_1,tg_2, tg_3} and the second partition containing the test goals {tg_4, tg_5, tg_6}.

Distributed Domination-based Partitioning. The distributed domination-based partitioning tries to create equally complex partitions. To this end, the sum of the post-order IDs of each test goal within a partition should be the same for all partitions.

> **Example 3.6.** Again, we will use partitions with the size of three test goals per partition. For distributed domination-based partitioning, we first order the test goals based on their post-order id in a descending manner. Afterwards, the test goals are grouped into the partition for alternating ascending and descending partition indexes. Therefore, in this example, the test goal with the highest post-order id (i.e., tg_1) is added to the first partition. The test goal with the second highest post-order id (i.e., tg_2) is added to the second partition. The next test-goal ((i.e., tg_3)) is also added to the second partition. Test goal tg_4 is then added to the first partition, and so forth. This leads us to the partition result with the first partition containing the test goals {tg_1,tg_3, tg_5} and the second partition containing the test goals {tg_2, tg_4, tg_6}.

3.2.1.5 *Global Information-based Partitioning*

In contrast to local information-based, global information-based partitioning also takes additional information of the CFA into account. The disadvantage, in this case, is the higher CPU time needed to perform the partitioning. However, if the information reduces the CPU time needed for the reachability analysis, the trade-off might be well worth it. Therefore, global information-based partitioning tries to use information about similarities between test goals to group similar test goals in the same partition. As these partitioning strategies are based on thresholds, they do not use a predefined number of partitions. Instead, the number of partitions is created on-demand.

Common-Path Partitioning. For the common-path partitioning for each test goal the shortest path to the start of the CFA is computed. The test goals are grouped in the same partition, if they share (at least partial) the same path. In contrast to the local information-based partitioning strategies introduced before,

this strategy does not use a predefined partition size, as the number of shared paths is unknown beforehand. This means new partitions are created on-demand, depending on the similarity of the test goals. First, a random test goal is selected, and the shortest path from the initial node of the CFA to the test goal is computed. Each test goal that is also present on this path is grouped in the same partition, and these test goals are removed from the set of test goals for partitioning. This means, no test goal is present in multiple partitions at the same time. This step is repeated until all test goals are split into partitions. Additionally, if a path is created that fully contains a previously created path, the partitions for both paths are merged. The advantage of this partitioning strategy is the shared paths between the test goals of the same partition and the efficiency of the partitioning. Although it is not as fast as local information-based partitioning, path computation is still quite efficient and, therefore, does not consume a large amount of CPU time. However, since only test goals with common paths are grouped in the same partition, the number of partitions might end up large. In this case, many reachability analyses are necessary, which might reduce efficiency.

Example 3.7. First, a random test goal is taken from the set of test goals. Consider tg_1 to be taken first. In this case, first, we will compute a path from the successor node of the test goal (i.e., node 4) to the initial CFA node. Figure 3.4a shows the corresponding path of the test goal. Next, we check if additional test goals are present on this path. As this is not the case, only test goal tg_1 is added to the first partition. This step is repeated again. In case of test goal tg_3 as next test goal, the computed path is equal to the path shown in Fig. 3.4b. As test goal tg_2 is present on this path as well, the second partition will contain the test goals tg_2 and tg_3.

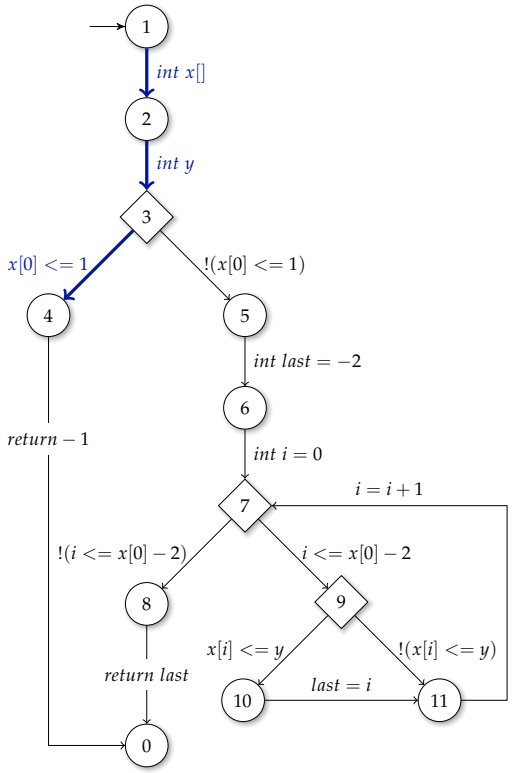
In case of tg_4 as next test goal for partitioning, the path would be as shown in Fig. 3.4c without the edge to node 10. Since test goal tg_2 is already in a previous partition, the only test goal for the third partition will be test goal tg_4.

In case of tg_5 as next test goal, the path will be equal to the path shown in Fig. 3.4c. As this path fully subsumes the previous path of test goal $tg4$, both partitions are merged. Therefore, the newly created partition number three will contain the test goals tg_4 and tg_5.

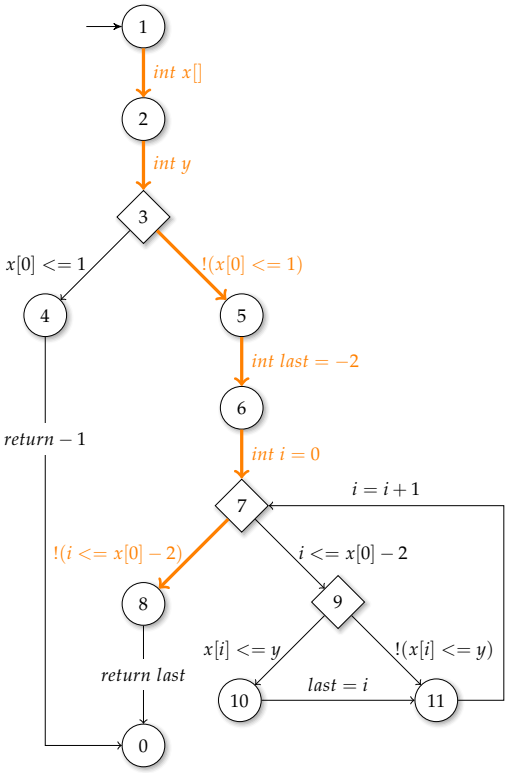
Lastly, the path for tg_6 is computed as shown in Fig. 3.4d. This test goal is also the only non-partitioned test goal on the path and, therefore, the only test goal added to partition number four. The final partitioning will be as follows.

- $partition_1$: $\{tg_1\}$
- $partition_2$: $\{tg_2,tg_3\}$
- $partition_3$: $\{tg_4,tg_5\}$
- $partition_4$: $\{tg_6\}$

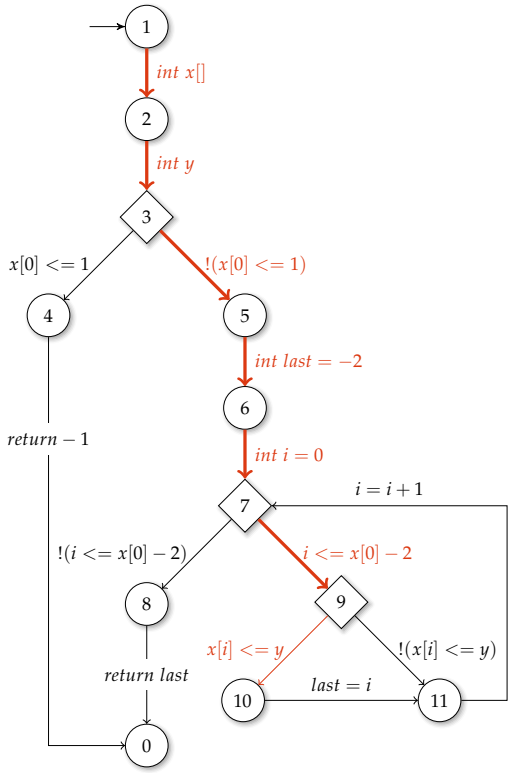
Common-Variable Partitioning. The common-variable partitioning strategy also computes paths from the test goal to the initial CFA node. However, this



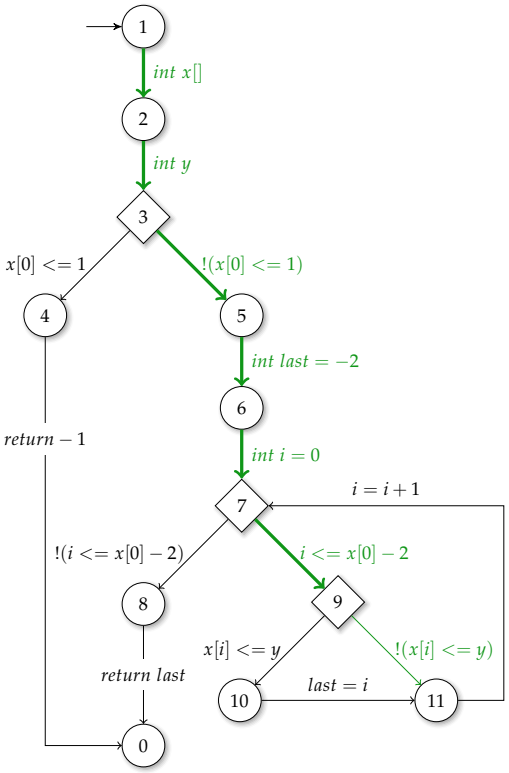
(a) First Path



(b) Second Path



(c) Third Path



(d) Fourth Path

Figure 3.4: CFA Paths

strategy only takes the variables used on these paths into account. Again, the partition size is not predefined for this partitioning strategy but depends on the similarity of the variables on the paths. First, for each test goal, the shortest path to the initial CFA node is computed. Next, all variables used on this path (either by assignment or reading access, however, initialization of variables are ignored) are collected and stored. Second, the similarity between test goals is computed using the following equation.

$$Similarity(tg_x, tg_y) = \frac{|Variables_{tg_x} \cap Variables_{tg_y}|}{|Variables_{tg_x} \cup Variables_{tg_y}|} \tag{3.1}$$

Lastly, all test goals with a similarity exceeding a predefined threshold are grouped in a common partition. Again, a test goal will never be present in multiple partitions. If a test goal could be grouped in multiple partitions, the first partition (depending on the order of the test goal creation) is taken.

Example 3.8. First, for each test goal, a path is computed. The paths for the test goals in our running example are shown in Fig. 3.4. Next, for each path, the used variables are stored. In this case, the variables used on the paths of the test goals are as follows.

- tg_1: $\{x\}$
- tg_2: $\{x\}$
- tg_3: $\{x, i\}$
- tg_4: $\{x, i\}$
- tg_5: $\{x, i, y, last\}$
- tg_6: $\{x, i, y, last\}$

Next, depending on a predefined threshold, each test goal is grouped in a partition. For each partition, all test goals' similarity value exceeds the threshold. Therefore, in the case of a threshold value of 1.0 only test goals which use exactly the same variables are grouped in the same partition. In this case, the result of the partitioning would be the following partitions.

- $partition_1$: $\{tg_1,tg_2\}$
- $partition_2$: $\{tg_3,tg_4\}$
- $partition_3$: $\{tg_5,tg_6\}$

3.2.2 *Hybrid Model Checking.*

One of the disadvantages of the previously illustrated test-case generation technique is the sole reliance on predicate analysis (i.e., usage of the PredicateCPA) for reachability analysis. While the predicate analysis is powerful, it is also expensive (in terms of CPU time) for successor computation during reachability analysis.

Previous work has already shown that in software verification for different analysis goals (e.g., reachability properties etc.), different analysis methods are more likely to compute a proof or counterexample within a given time limit. [36] Also, for test-case generation, the first successes to increase efficiency through multiple analysis techniques have been achieved [24, 25, 100]. Even though multi-property checking has already been utilized for these techniques, partitioning is still out of scope for those works.

Therefore, to further increase efficiency for our test-case generation approaches, we combined techniques for hybrid model checking with our partitioning techniques.

Figure 3.5 shows the overview of the hybrid model checking methodology combined with partitioning. First, the source code is parsed and translated into a CFA. Next, the test goals are derived from the CFA based on a coverage criterion. Afterwards, different reachability analyses are executed successively. For each analysis, first, all uncovered test goals are partitioned. Afterwards, the reachability analysis is started with a predefined time limit. After the time limit, the remaining uncovered test goals are again partitioned, and the next analysis is started. For each analysis, a different partitioning strategy may be defined to optimize efficiency even further. If the global time limit is not exceeded after executing the last analysis (and there exist more uncovered test goals), the first analysis is started again, and the process starts all over again. Additionally, if a test goal is proven to be unreachable by any of the analyses, the test goal is removed from the set of uncovered test goals as well.

Analysis Techniques. For reachability analysis, this thesis utilizes a predicate analysis, as well as explicit value analysis. As described before, the predicate analysis is quite powerful and is often able to summarize loops easily. However, it can be rather slow due to expensive SMT-Solver calls [24]. In contrast, the explicit value analysis is fast while tracking few variables. However, in case a test goal depends on many variables, the explicit value analysis slows down significantly. Additionally, the explicit value analysis might get stuck in loops and suffers from a large state space if many assignments are present [24]. This enables the analysis to reach test goals which might be complex to reach by the predicate analysis rather fast, as long as the amount of variables the test goal depends on is small.

Remarks. While it is also possible, to combine different test-case generation techniques in general (e.g., combining symbolic execution and model checking, see [28]), we will focus on different analysis techniques for model checking in the course of this thesis.

3.3 Implementation

To evaluate the illustrated techniques, we implemented them in the CPAchecker framework in the course of this thesis.

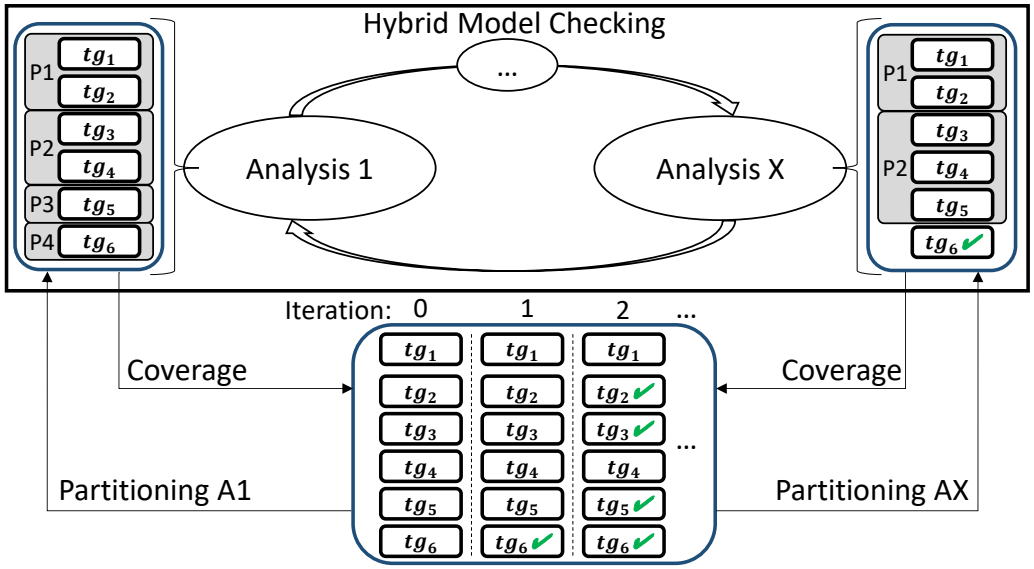


Figure 3.5: Hybrid Test-Case Generation Overview [23]

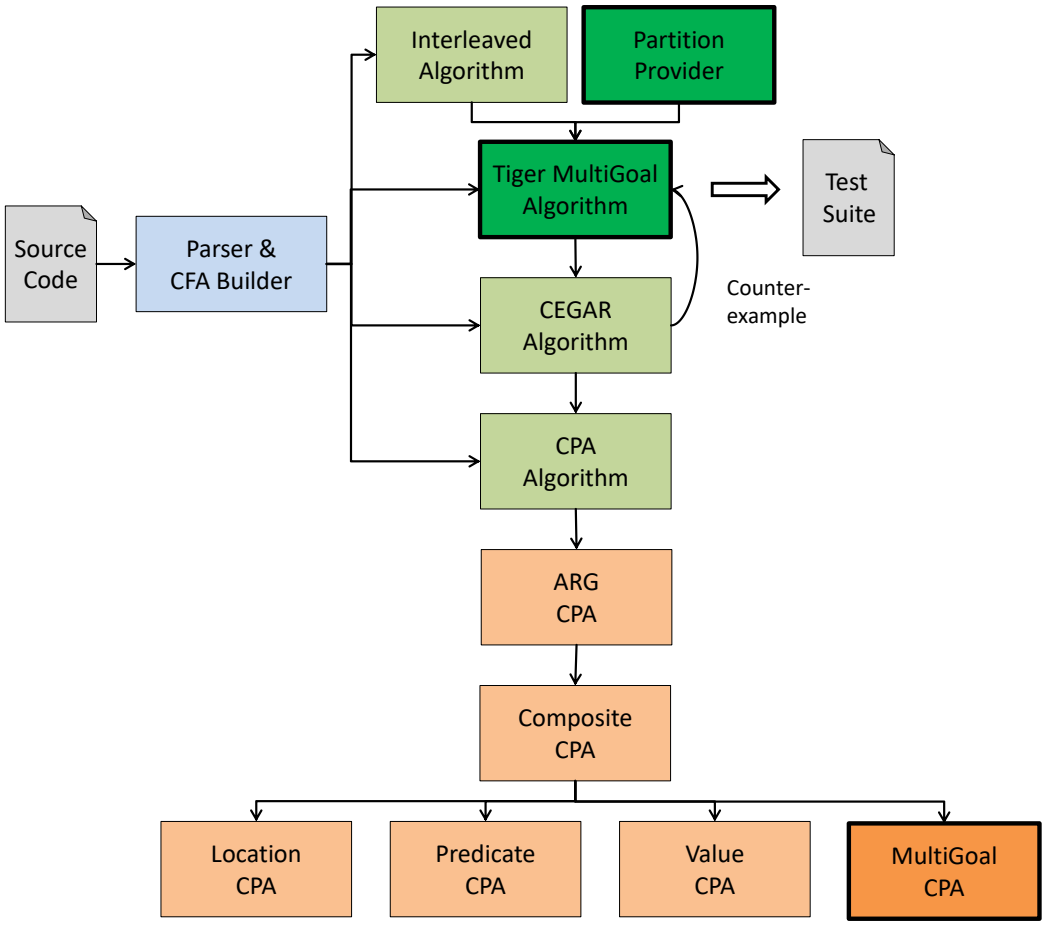


Figure 3.6: Modules for Test-Case Generation with CPAchecker (adapted from [21])

Figure 3.6 shows the additional modules. Each module with a thick border is either new or changed. The new and changed modules are explained in the remainder of this section.

3.3.1 *Tiger Multigoal Algorithm*

The original Tiger Algorithm is now enhanced to support partitioning and multi-property checking. Another additional functionality is the sanitation of the ARG whenever a test goal is covered. As the reachability analysis is now resumed, if there are still uncovered test goals in the partition, the target state corresponding to the newly found test goal needs to be removed. Additionally, weaved CFA edges, and their successor and predecessor ARG states need to be removed. Otherwise, the reachability analysis would lead to incorrect results for the remaining test goals. As shown in Fig. 3.1, the ARG state $7'$ forces counterexamples to only take the left path (due to the weaved constraint). Therefore, each test goal that needs to take the right path would be unreachable and, therefore, lead to incorrect results. Removing these states from the ARG enables the reachability analysis to resume as intended, as the test goal leading to the weaved edges is not part of the partition after being covered. Therefore, these edges will not be weaved in the ARG again during this reachability analysis.

The remaining weaved edges (e.g., state $9'$ or $11'$ Fig. 3.1) will not be removed, as this would lead to expensive recomputation of the following states. Additionally, the weaved edges will not lead to any undesired behavior and will be removed if a new CEGAR refinement is executed anyway.

3.3.2 *MultiGoalCPA*

The MultiGoalCPA now replaces the old TestgoalCPA. The MultiGoalCPA uses a set of test goals, each test goal consisting of a list of CFA edges.

The abstract state of the MultiGoalCPA (called MultiGoalState) contains a set of tuples to track the progress of test goals. The tuples consist of the test goal as key and an integer as value to track the next edge of the test goal that needs to be passed during reachability analysis.

For the initial abstract state, for each test goal consisting of multiple CFA edges, a new variable is weaved into the ARG to force the counterexample later to cover the test goal (see Sect. 3.1). The MultiGoalState carries the information that should be weaved into the ARG. However, it is not responsible for the actual weaving. This is handled by the strengthening operator of the LocationCPA and is explained in the next subsection.

The transfer relation computes a successor state by checking for each test goal if the current edge to compute the successor equals the next edge needed for the test goal. In this case, the value of the tuple is incremented. If the test goal consists of multiple edges, the corresponding weaved variable for this test goal is incremented by weaving an additional CFA edge into the ARG.

If the test goal is covered (i.e., all edges of the test goal have been traversed), the last constraint for the test goal is weaved if the test goal consists of multiple edges.

After traversing the weaved edge (or if the test goal consisted of only one edge), the MultiGoalState is set as target state, and a counterexample is computed.

3.3.3 *LocationCPA*

As the CFA cannot be modified during runtime, the information weaving must be done during the reachability analysis by modifying the ARG.

If weaving is needed (which is triggered by the MultiGoalState), the current edge for successor computation is copied, and a new successor CFA node is created. Additionally, the successor CFA node receives a new leaving edge that is responsible for weaving the new information. The successor for the weaved edge is the original successor of the current abstract state (see Fig. 3.1).

Additionally, caching of weaved CFA edges is crucial. In the case of a CEGAR refinement, the information about unreachable parts of the ARG is sometimes based on the instantiations of CFA edges. Therefore, if a test goal is not reachable due to the newly weaved edges, the edges need to be the same objects as before the refinement. Otherwise, CEGAR will endlessly refine the reachability analysis.

3.3.4 *Partitioning*

To implement the new partitioning strategies, a new module was introduced to CPAchecker called PartitionProvider. The desired partition strategy, size, and other parameters can be adjusted via the CPAchecker internal configuration functionality. Next, the set of test goals derived from the CFA can be passed as parameters and will be returned as partitions, depending on the selected strategy.

3.3.5 *Hybrid Model Checking*

For utilizing hybrid model checking, the interleaved algorithm provided by CPAchecker is used. Additionally, we use the ValueCPA for explicit value analysis.

Lastly, we upgraded the Tiger MultiGoal Algorithm to store derived test goals and test cases throughout the full test-case generation process. This is necessary, as each reachability analysis instantiates a new *Tiger MultiGoal Algorithm* object.

3.4 Evaluation

The previously introduced partitioning strategies and the hybrid model checking technique allows for adjustable configurations for test-case generation. In our experimental evaluation, we consider C programs as systems under test. These C programs must be self-contained (i.e., no external function calls or external global variables, except for test input values). Based on this experimental setting, we will investigate the impact of our strategies for efficiency and effectiveness.

3.4.1 *Research Questions.*

- **RQ1** Which test-case generation strategy has the strongest impact on the effectiveness of the resulting test-suites in terms of achieved coverage?
  - **RQ1.1** Which partitioning strategy has the strongest impact on the effectiveness of the resulting test-suites in terms of achieved coverage?
  - **RQ1.2** Which hybrid model checking strategy has the strongest impact on the effectiveness of the resulting test-suites in terms of achieved coverage?
- **RQ2** Which test-case generation strategy has the strongest impact on the efficiency of the resulting test-suites?
  - **RQ2.1.1** Which partitioning strategy has the strongest impact on the efficiency on the test-case generation in terms of CPU time.
  - **RQ2.1.2** Which partitioning strategy has the strongest impact on the efficiency of the resulting test-suites in terms of test-suite size.
  - **RQ2.2.1** Which hybrid model checking strategy has the strongest impact on the efficiency on the test-case generation in terms of CPU time?
  - **RQ2.2.2** Which hybrid model checking strategy has the strongest impact on the efficiency of the resulting test-suites in terms of test-suite size?

3.4.2 *Experimental Setup*

Next, we will describe the evaluation methods and experimental design used in our experiments to address the research questions.

Methods and Experimental Design. To evaluate the partitioning strategy and the partition size, we will focus on the following strategies. The strategy names are based on the partitioning configuration. PS_9: [Dom, 25T] uses *Dominiation* as partitioning strategy with a partition size of 25 test goals ($T->$ *total*, $P->$ *percentage*). Strategies with [RevDom,...] use *reversedomination*-based and strategies with [DistDom,...] use *distributed domination*-based partitioning.

- PS_1: [-,100P]
- PS_2: [ComPath, -]
- PS_3: [ComVar, -]
- PS_4: [DistDom, 25P]
- PS_5: [DistDom, 25T]
- PS_6: [DistDom, 50P]
- PS_7: [DistDom, 50T]
- PS_8: [Dom, 25P]
- PS_9: [Dom, 25T]
- PS_{10}: [Dom, 50P]
- PS_{11}: [Dom, 50T]
- PS_{12}: [REvDom, 25P]
- PS_{13}: [REvDom, 25T]
- PS_{14}: [REvDom, 50P]
- PS_{15}: [REvDom, 50T]

Therefore, we will evaluate 15 different strategies for partitioning to research the impact on effectiveness and efficiency. Additionally, we will use strategy PS_1 (standard multi-property checking) as a baseline to compare the impact of the remaining strategies. Those strategies will be evaluated for both the predicate analysis (utilizing the PredicateCPA) and the explicit value analysis (utilizing the ValueCPA). Next, we will use the respectively best strategy in terms of effectiveness (on average) to further evaluate different hybrid model checking strategies. In this case, we will evaluate different orders of analyses and different time limits for the analyses.

The following strategies show the different time limits based on the global time limit and their corresponding analysis technique (i.e., HS_1: [450V,450P] will run the explicit value analysis first for 450 seconds. Afterward, the predicate analysis is executed for 450 seconds).

- HS_1: [450V, 450P]
- HS_2: [225V, 675P]
- HS_3: [120V, 780P]
- HS_4: [675V, 225P]
- HS_5: [780V, 120P]
- HS_6: [450P, 450V]
- HS_7: [225P, 675V]
- HS_8: [120P, 780V]
- HS_9: [675P, 225V]
- HS_{10}: [780P, 120V]

Subject Systems. For evaluating our strategies, we use a selection of C programs from the SV-Benchmarks[1] corpus. SV-Benchmarks contains real-world and crafted C programs used for evaluating software verification (SV-Comp[2]) and test-case generation techniques (Test-Comp[3]). We selected a representative selection of C programs from crafted as well as real-world programs. Our selected C programs contain at least 100 branches and consist of 535 to 6894 lines of code. The list of programs can be found in Appendix A.

Data Collection. For each program of our subject systems, we apply the test-case generation approach using all strategies. As coverage criterion, we selected branch coverage. To answer our research questions we first generate a test suite T and measure the CPU time using benchexec [4]. Benchexec is a framework for reliable benchmarking, enabling us to enforce time limits and accurately measure CPU time. Next, to answer **RQ1** measure the actual coverage (i.e., the number of test goals reached) by executing the test suite on the program under test utilizing the test-suite executor TestCov[5].

$$\mathit{effectiveness}(T) = \frac{\sum_{i=1}^{z} \mathit{reached}(tg_i, T)}{z} \tag{3.2}$$

[1] https://github.com/sosy-lab/sv-benchmarks
[2] https://sv-comp.sosy-lab.org/2021/
[3] https://test-comp.sosy-lab.org/2021/
[4] https://github.com/sosy-lab/benchexec/
[5] https://gitlab.com/sosy-lab/software/test-suite-validator

where $reached(tg_i, T) = 1$ if the test goal is reached by any test case in test suite T, or 0 otherwise, and z is the number of test goals. The effectiveness of a strategy s is calculated as

$$effectiveness(s) = \frac{\sum_{i=1}^{n} effectiveness(T_i^s)}{n} \tag{3.3}$$

where n is the number of programs under test (i.e., the subject systems) and T_i^s is the test suite generated for subject system number i with the strategy s. Note, that TestCov also includes unreachable test goals in the measurement, as it cannot differentiate between reachable and unreachable test goals. Therefore, it is often not possible to cover 100% of the goals (for example test goals corresponding to dead code).

Finally, to answer **RQ2**, the efficiency in terms of test-suite size of test suite (T) is calculated as

$$efficiency_{size}(T) = |T| \tag{3.4}$$

where $|T|$ is the number of test cases of the test suite. Based on this equation, the overall efficiency of strategy s is calculated as

$$efficiency_{size}(s) = \frac{\sum_{i=1}^{n} efficiency_{size}(T_i^s)}{n}. \tag{3.5}$$

The efficiency in terms of CPU time of a strategy s is calculated as

$$efficiency_{cpu}(s) = \frac{\sum_{i=1}^{n} efficiency_{cpu}(T_i^s)}{n}. \tag{3.6}$$

where $efficiency_{cpu}$ is the CPU time needed to generate test suite T_i^s.

As we measure CPU time instead of wall time, run-time fluctuations are negligible. Additionally, running a specific strategy multiple times will lead to the same results. Therefore, strategies are evaluated once.

Measurement Setup. Our test-case generation tool has been executed on an Ubuntu 18.04 machine, equipped with an Intel Core i7-7700k CPU and 64 GB of RAM. The CPU time for test-suite generation is limited to 900 s for each program under test, and the CPU time for the execution of the test cases to detect bugs is limited to 30 s per test case. We executed our evaluation with Java 1.8.0-171 and limited the Java heap to 6 GB.

3.4.3 *Results.*

Fig. 3.7 shows the results of our experimental evaluation concerning research question **RQ1**. The results of our evaluation regarding **RQ2** are shown in Fig. 3.8 and Fig. 3.9.

RQ1.1 Effectiveness of Partitioning. Figure 3.7a shows the effectiveness of the partitioning strategies for test-case generation with predicate analysis. The figure shows that most partitioning strategies lead to lower effectiveness compared to $[-, 100P]$ (i.e., all test goals in a single reachability analysis). Only the

strategy $[Dom, 50P]$ leads to higher effectiveness and, therefore, to the highest effectiveness of test-case generation with the predicate analysis.

Figure 3.7b shows the effectiveness of the partitioning strategies for test-case generation with explicit value analysis. The figure shows that most partitioning strategies lead to higher effectiveness compared to $[-, 100P]$ (i.e., all test goals in a single reachability analysis). The highest effectiveness is obtained by the strategy $[ComPath, -]$.

RQ1.2 Effectiveness of Hybrid Model Checking. Figure 3.7c shows the effectiveness of the hybrid model checking strategies with partitioning strategy $[-, 100P]$. The figure shows that the highest effectiveness is reached by starting the analysis with 120 seconds for the explicit value analysis and afterward 780 seconds for the predicate analysis.

Figure 3.7d shows the effectiveness of the hybrid model checking strategies with partitioning strategy $[Dom, 25T]$ for the explicit value analysis and $[Dom, 50P]$ for the predicate analysis. The highest effectiveness is achieved by the strategies $[675V - 225P]$ and $[120V - 780P]$.

RQ2.1.1 $Efficiency_{CPU}$ of Partitioning. Figure 3.8a shows the results regarding $efficiency_{CPU}$ of the partitioning strategies for test-case generation with predicate analysis, and Fig. 3.8b shows the $efficiency_{CPU}$ of the partitioning strategies for test-case generation with the explicit value analysis

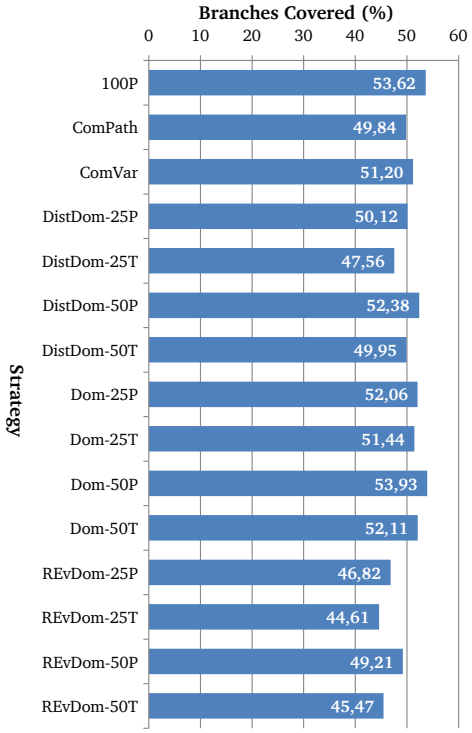
The figures show that for the predicate analysis and the explicit value analysis, most partitioning strategies lead to higher $efficiency_{CPU}$ (i.e., lower CPU time) compared to $[-, 100P]$. The highest $efficiency_{CPU}$ for the predicate analysis constitutes partitioning strategy$[ComVar, -]$ and for the explicit value analysis $[RevDom, 25T]$.

RQ2.1.2 $Efficiency_{Size}$ of Partitioning. Figure 3.9a shows the $efficiency_{size}$ of the partitioning strategies for test-case generation with predicate analysis. The figure shows that all partitioning strategies lead to higher $efficiency_{size}$ (i.e., less test cases) compared to $[-, 100P]$. The highest $efficiency_{size}$ constitutes partitioning strategy $[RevDom, 50T]$.

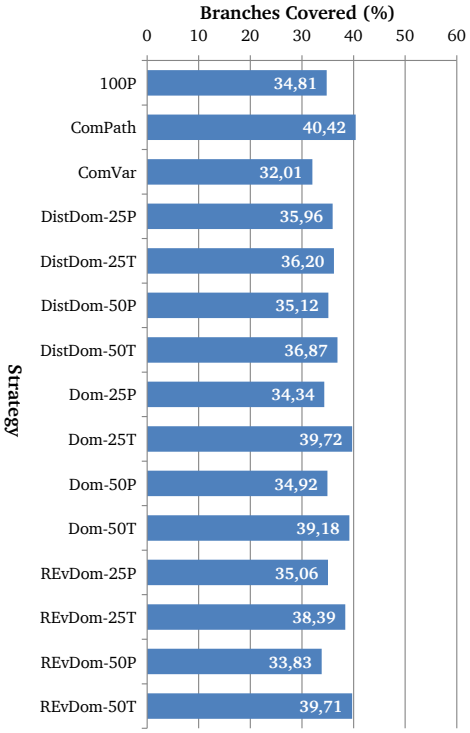
Figure 3.9b shows the $efficiency_{size}$ of the partitioning strategies for test-case generation with explicit value analysis. The figure shows that most partitioning strategies lead to lower $efficiency_{size}$ (i.e., more test cases) compared to $[-, 100P]$. The highest $efficiency_{size}$ constitutes partitioning strategy $[ComVar, -]$ and the lowest $efficiency_{size}$ is obtained by partitioning strategy $[RevDom, 50T]$.

RQ2.2.1 $Efficiency_{CPU}$ of Hybrid Model Checking. Figure 3.8c shows the results regarding $efficiency_{CPU}$ of the hybrid model checking strategies with partitioning strategy $[-, 100P]$. The figure shows that the best hybrid strategy for $efficiency_{CPU}$ is $[780P, 120V]$.

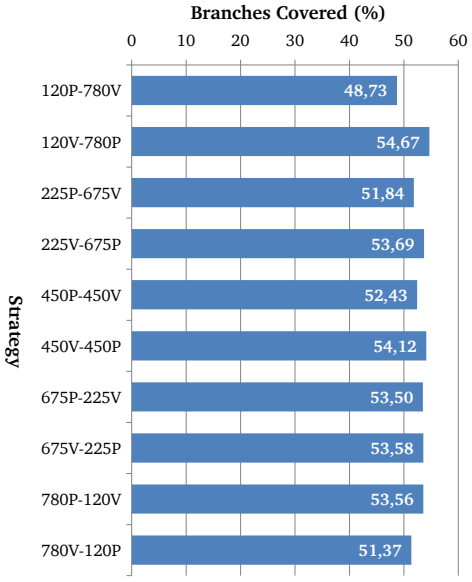
Figure 3.8d shows the $efficiency_{CPU}$ of the hybrid model checking strategies with partitioning strategy $[Dom, 25T]$ for the explicit value analysis and $[Dom, 50P]$ for



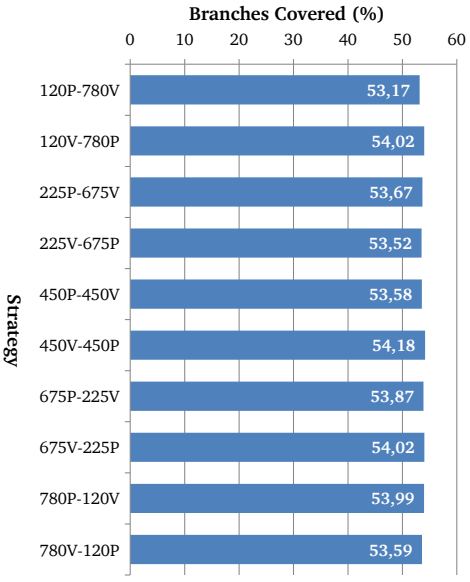
(a) Branch Coverage for the Predicate Analysis



(b) Branch Coverage for the Explicit Value Analysis



(c) Branch Coverage for Hybrid Model Checking with Partitioning Strategy $[-, 100P]$



(d) Branch Coverage for Hybrid Model Checking with best Partitioning Strategies

Figure 3.7: Results Effectiveness

the predicate analysis. The figure shows that the best hybrid strategy in terms of $efficiency_{CPU}$ is $[225V, 675P]$.

RQ2.2.2 $Efficiency_{Size}$ of Hybrid Model Checking. Figure 3.9c shows the results regarding $efficiency_{size}$ of the hybrid model checking strategies with partitioning strategy $[-, 100P]$. The figure shows that the best hybrid strategy in terms of $efficiency_{size}$ is $[120P, 780V]$.

Figure 3.9d shows the $efficiency_{size}$ of the hybrid model checking strategies with partitioning strategy $[Dom, 25T]$ for the explicit value analysis and $[Dom, 50P]$ for the predicate analysis. The figure shows that the best hybrid strategy in terms of $efficiency_{size}$ is $[225V, 675P]$.

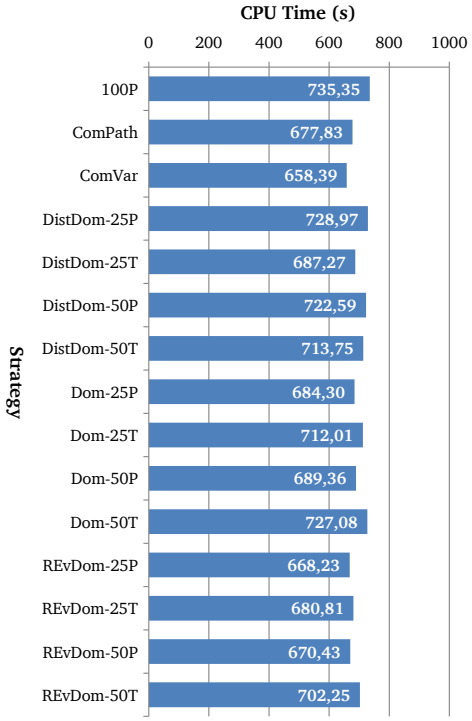
3.4.4 *Discussion.*

RQ1.1 Effectiveness of Partitioning. For the predicate analysis, only one partitioning strategy (i.e., $[Dom, 50P]$) was able to improve effectiveness by approx. 0.31% on average. However, for specific subject systems, the different partitioning strategies impact effectiveness by more than 15% compared to partitioning strategy $[-, 100P]$. Therefore, the choice of the partitioning strategy should depend on the subject system itself. For the explicit value analysis, most partitioning strategies lead to higher effectiveness compared to partitioning strategy $[-, 100P]$. The best effectiveness is achieved by partitioning strategy $[ComPath, -]$ and improves effectiveness compared to strategy $[-, 100P]$ by 5.61%. The difference in effectiveness of single subject systems is even more significant for the explicit value analysis. For specific subject systems, the difference compared to strategy $[-, 100P]$ is up to 53.50%. We assume the significant impact of the partitioning strategy with the explicit value analysis is due to the fact that the explicit value analysis is often able to cover test goals either very fast or extremely slow. Therefore, without partitioning or by choosing sub-optimal partitioning strategies, the explicit value analysis is more likely trying to cover a test goal that takes a long time and, therefore, will run into the timeout before covering the easier test goals.

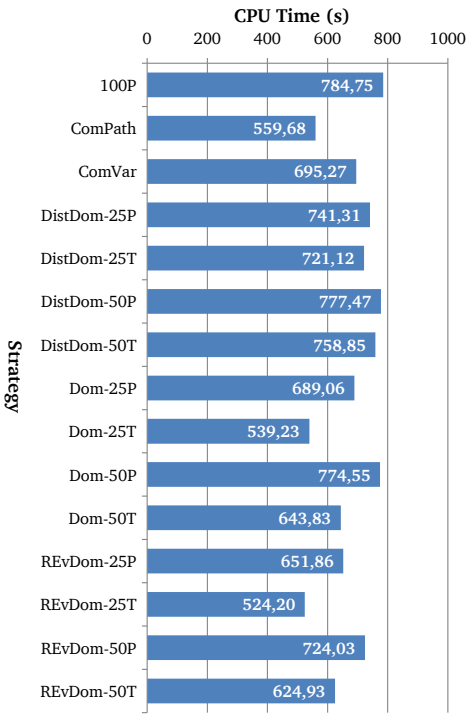
For both analysis techniques, the choice of the partitioning strategy impacts the effectiveness by up to approx. 10% for branch coverage.

RQ1.2 Effectiveness of Hybrid Model Checking. Without partitioning (i.e., strategy $[-, 100P]$), the best effectiveness is achieved by hybrid strategy $[120V, 780P]$. However, there is no apparent impact of effectiveness when changing the partial time limit of the explicit value analysis. The reason for this behavior is most likely that the explicit value analysis is able to cover test goals either very fast or extremely slow. Therefore, the number of test goals covered does not differ significantly for different partial time limits. However, the results show that executing the explicit value analysis before the predicate analysis always leads to an improvement in effectiveness (for the same partial time limits).

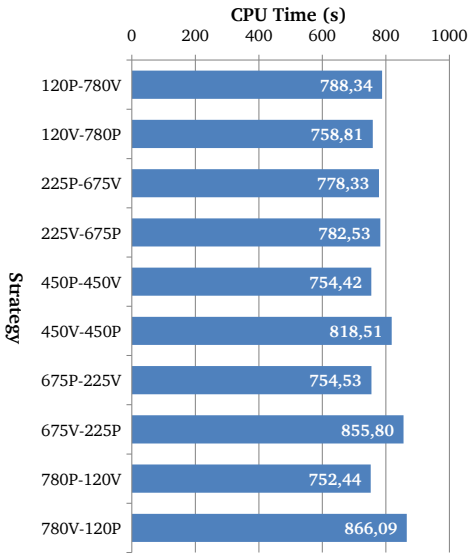
Using the partitioning strategies that performed well for the corresponding analyses (i.e., $[Dom, 50P]$ for the predicate analysis and $[Dom, 25T]$ for the ex-



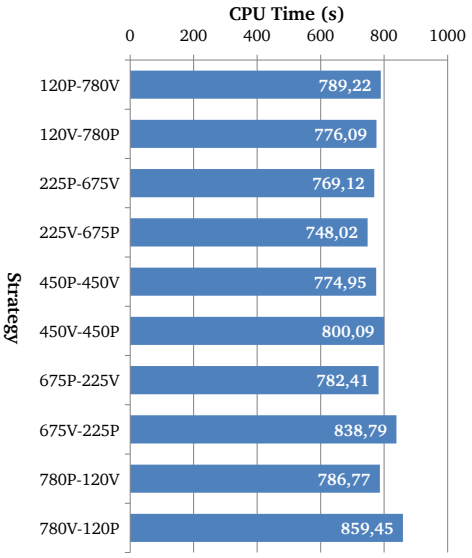
(a) $Efficiency_{CPU}$ for the Predicate Analysis



(b) $Efficiency_{CPU}$ for the Explicit Value Analysis

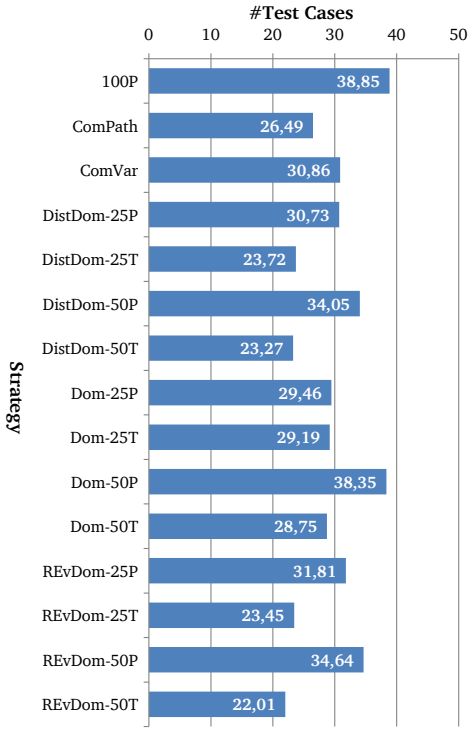


(c) $Efficiency_{CPU}$ for Hybrid Model Checking with Partitioning Strategy $[-, 100P]$

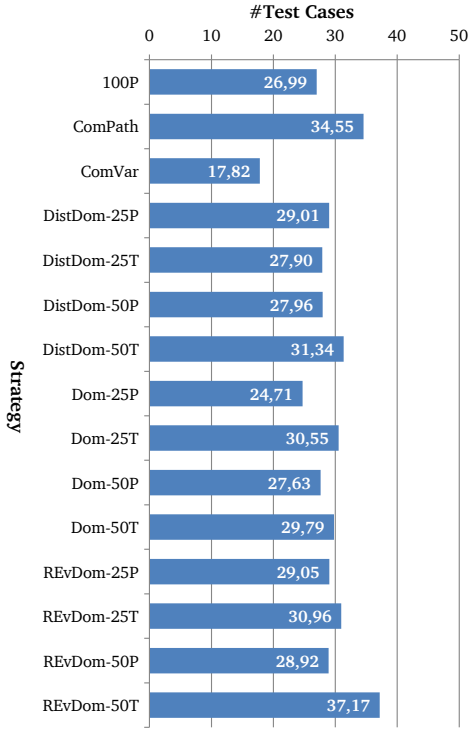


(d) $Efficiency_{CPU}$ for Hybrid Model Checking with best Partitioning Strategies

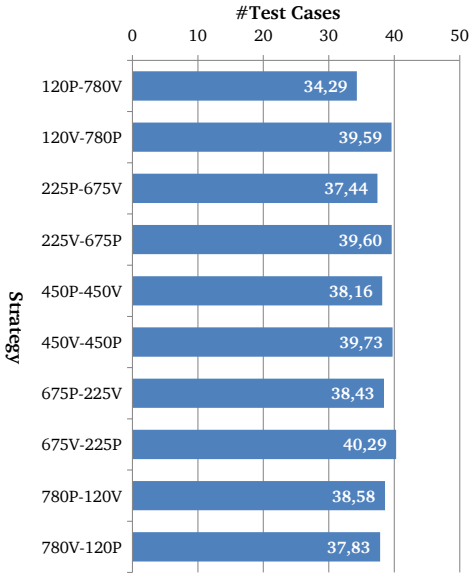
Figure 3.8: Results $Efficiency_{CPU}$



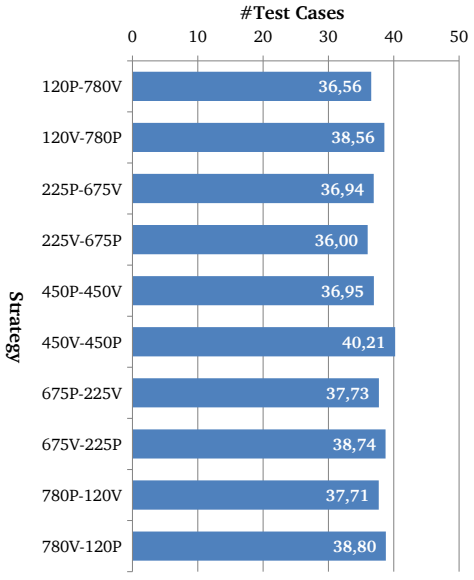
(a) $Efficiency_{Size}$ for the Predicate Analysis



(b) $Efficiency_{Size}$ for the Explicit Value Analysis



(c) $Efficiency_{Size}$ for Hybrid Model Checking with Partitioning Strategy $[-, 100P]$



(d) $Efficiency_{Size}$ for Hybrid Model Checking with best Partitioning Strategies

Figure 3.9: Results $Efficiency_{Size}$

plicit value analysis) increases the effectiveness for all but one hybrid strategy. We chose the partitioning strategy $[Dom, 25T]$ instead of $[ComPath, -]$ for the explicit value analysis for practical reasons. Only the hybrid strategy $120V - 780P$ leads to a decrease in effectiveness by using the other partitioning strategies. The hybrid strategy leading to the best effectiveness for those partitioning strategies is $[450V, 450P]$. Although, the effectiveness differs only slightly for other strategies. However, this leads to a decrease in the effectiveness of the best hybrid strategy compared to hybrid strategies with partitioning strategy $[-, 100P]$. This is most likely due to the fact that partitioning improves the explicit value analyses by a large margin and, therefore, leads to better results for longer partial time limits for the explicit value analysis. However, the hybrid strategies perform better for longer partial time limits for the predicate analysis. Therefore, the overall strategies perform similarly in terms of effectiveness. Additionally, for specific subject systems, partitioning does improve effectiveness. Therefore, the choice of the partitioning strategy should depend on the subject system itself for hybrid model checking, as well. However, for the hybrid strategies, the subject system does not influence the result by a large margin. The best strategy (i.e., $[120V - 780P]$) leads to the best effectiveness for nearly all subject systems. Therefore, the optimal choice of the hybrid strategy does not depend on the subject system itself.

RQ2.1.1 $Efficiency_{CPU}$ OF PARTITIONING. The choice of the partitioning strategy for the predicate analysis does affect $efficiency_{CPU}$. However, there is no apparent correlation between the number of partitions and the CPU time needed. Additionally, strategy leading to higher effectiveness does not necessarily lead to higher or lower $efficiency_{CPU}$.

For the explicit value analysis, the number of partitions does affect the efficiency in terms of CPU time. Partitioning strategies with higher partitions lead to higher $efficiency_{CPU}$ (i.e., lower CPU time). Additionally, most strategies leading to higher effectiveness also lead to higher $efficiency_{CPU}$.

RQ2.1.2 $Efficiency_{Size}$ OF PARTITIONING. The choice of the partitioning strategy for the predicate analysis has a strong effect on $efficiency_{size}$. However, the $efficiency_{size}$ does seem to correlate with the effectiveness (i.e., higher effectiveness leads to lower $efficiency_{size}$). Also, there is no apparent correlation between the number of partitions and the CPU time needed. Additionally, strategies leading to higher effectiveness do not necessarily lead to higher or lower $efficiency_{CPU}$.

The choice of the partitioning strategy for the explicit value analysis has a strong effect on $efficiency_{size}$. Additionally, there seems to be a correlation between both the partition size and the partitioning technique. Interestingly, the strategy achieving the best effectiveness (i.e., $[ComPath, -]$) is not the strategy with the worst $efficiency_{size}$. Compared to strategy $[RevDom, 50T]$ strategy $[ComPath, -]$ improves both $efficiency_{size}$ and effectiveness.

RQ2.2.1 $Efficiency_{CPU}$ OF HYBRID MODEL CHECKING. The choice of the partial time limits for hybrid model checking with partitioning strategy $[-, 100P]$ does affect $efficiency_{CPU}$. It does correlate with the partial time limit of the explicit

value analysis (i.e., longer partial time limits for the explicit value analysis leads to lower $efficiency_{CPU}$) if the explicit value analysis is executed first. If the predicate analysis is executed first, the $efficiency_{CPU}$ stays similar to different partial time limits. The reason for this behavior is most likely that the explicit value analysis is able to cover test goals that are hard to cover for the predicate analysis. Therefore, executing the explicit value analysis first reduces the time needed for the predicate analysis. However, this does not seem to be the other way around as well.

Using the partitioning strategies that performed best for the corresponding analyses, the $efficiency_{CPU}$ is affected by choice of the partial time limits. For lower partial time limits for the first analysis, the $efficiency_{CPU}$ increases when executing the explicit value analysis first. This changes for longer partial time limits for the first analysis. In this case, starting the predicate analysis first increases $efficiency_{CPU}$.

RQ2.2.2 $Efficiency_{CPU}$ of Hybrid Model Checking. The choice of the partial time limits for hybrid model checking with partitioning strategy $[-, 100P]$ does affect $efficiency_{size}$. However, it does seem to correlate with partial time limits. Although, the $efficiency_{size}$ does seem to correlate with the effectiveness (i.e., higher effectiveness leads to lower $efficiency_{size}$).

Using the partitioning strategies that performed best for the corresponding analyses, the $efficiency_{size}$ is affected by the choice of the partial time limits. However, there seems to be no direct correlation with the partial time limits or the effectiveness of the hybrid strategy.

Interestingly, the partitioning strategies have been shown to lead to even higher effectiveness compared to $[-, 100P]$ in previous works (e.g., student theses). However, (most likely) due to improvements to the CPAchecker framework and the implementation of the Tiger MultiGoal algorithm, the benefits of partitioning have been reduced. Nonetheless, the strategies provide a solid base for parallelization of test-case generation in future works.

3.4.5 *Threats to Validity*

Internal Validity. The technique for test-case generation based on hybrid model checking and partitioning for multi-property checking allows for adjustable settings to optimize effectiveness and efficiency. In our evaluation, we measured the impact of the partitioning strategy and size and different settings for hybrid model checking. We limited our evaluation to a predefined partition size and a predefined set of time limits for hybrid model checking to keep the evaluation graspable. Therefore, it might be possible we missed even better settings to improve test-case generation even further. However, as we compared our strategies to different baselines and improved effectiveness, missing even better solutions does not invalidate our results and only shows possible future work. Another threat to internal validity might be our choice of software model checking for test-case generation. At least for our subject systems, there exist tools that achieve higher effectiveness (VeriFuzz achieves 74.73% branch coverage for our subject

systems [6]). However, software model checking is often able to determine whether a test goal is unreachable, which is sometimes necessary (e.g., for Chapter 5). Therefore, we assume our choice does not threaten our internal validity.

Additionally, we might threaten our internal validity by our choice of branch coverage as a coverage criterion. However, branch coverage is one of the most prominent examples for coverage criteria and is also used in competitive test-case generation environments (see Test-Comp[7]). We also assume similar results for different coverage criteria based on our experience.

Finally, we expect our current limitation to C programs to also not seriously harm validity as we can expect similar results for other programming languages, at least for those relying on an imperative core language (e.g., most OO languages).

External Validity. We are not aware of other tools, which support the partitioning of test goals. Therefore, we cannot evaluate the partitioning strategies on other tools based on different test-case generation techniques (e.g., symbolic execution or fuzzing).

Another threat might arise from the selection of our subject systems. However, as we selected our subject systems from a corpus used for the software testing competition (Test-Comp), we assume our selection to be a valid basis for evaluating test-case generation techniques.

Finally, our evaluation relies on third-party software, namely the test-suite executor TestCov and Benchexec. However, TestCov is established and has been extensively used as a test-suite validator and coverage measurement in the last three years of Test-Comp. Benchexec is even more established, as it is also used for Test-Comp and additionally used in the last five years of SV-Comp. Therefore, we expect both tools to produce sound results.

3.5 Related Work

In this section, we will discuss related work on test-case generation and hybrid model checking. We are not aware of related works in terms of the partitioning of test goals to increase the performance of test-case generation. However, there are multiple related works on general test-case generation approaches, which we will group based on their test-case generation technique.

Fuzzing. Test-case generation with fuzzing is currently very popular in research. The main idea is to generate new test cases by generating (semi-)random input values. Sometimes the input values of existing test cases are used and only modified to generate new test cases. [119] There are many different fuzzing methods, for example, based on evolutionary algorithms [145] or based on context-free grammars [79]. In the last years, even greybox fuzzing [182] or whitebox fuzzing [80, 79] have been subject to research. However, since fuzzing is often not based on the test goals, partitioning of test goals is mostly irrelevant. There-

[6] https://test-comp.sosy-lab.org/2021/results/results-verified/
[7] https://test-comp.sosy-lab.org/2021/

fore, these works do not consider partitioning of test goals for optimizing test-case generation. For hybrid test-case generation, however, different work exists. For example, Verifuzz combines different fuzzing tools to optimize test-case generation further. [19]

Plain Random. A similar test-case generation approach, even though way simpler, to fuzzing is plain random test-case generation. [117] For plain random, a test case is fully randomly generated, and afterwards, the coverage for the system under test is measured. The advantage is the efficiency for test-case generation. However, complex test goals are often not reached, as the chances to generate the correct input values for the test case are slim (e.g., to create a test case which evaluates true for $x == 1$ is $\frac{1}{2^{32}}$ for a 32-bit system). For plain random test-case generation partitioning of test goals is irrelevant, as the basis for test-case generation are only randomly generated input values, not the test goals themselves.

Symbolic Execution. For symbolic execution, a symbolic reachability graph is generated (similar to the ARG). There exist different works and tools for the optimization of symbolic execution. One prominent example is Klee. [42] However, symbolic execution often does not build the symbolic reachability graph tailored for specific test cases. Due to this reason, there exists no work for partitioning of test goals for symbolic execution, as far as we know. However, recent work that combines symbolic execution with CEGAR [27] might lead to the possibility of partitioning test goals, as CEGAR allows for different abstraction based on the test goals. For hybrid test-case generation, different works try to combine symbolic execution with different techniques, for example, fuzzing. [177, 47]

Bounded Model Checking. Recent work also used bounded model checking for test-case generation. [75] For bounded model checking, program states are also computed (either abstract or concrete, depending on the technique). However, loops are only unrolled for a specific number of iterations k. This enables the model checker to prove or disprove properties that hold for at least k loop iterations. Therefore, this can be used to generate test cases in the same manner as symbolic model checking. However, it is restricted for test goals, which need no more than k loop iterations to be reached. However, for bounded model checking, there is no work that partitions test goals or uses hybrid test-case generation that we know of.

Software Model Checking. There also exists other work, which uses software model checking similarly to this thesis. CoVeriTest is also a test-case generator, also based on the CPAchecker framework. [100] While CoVeriTest also uses hybrid approaches, it does not use any partitioning of test goals, and, therefore, does not combine partitioning with hybrid model checking.

To conclude, there exist various works on test-case generation with different techniques, sometimes similar to our approach. However, none of these techniques combine partitioning of test goals with hybrid test-case generation. Additionally, for some coverage criteria (especially in Industrie 4.0) 100% coverage

is needed (e.g., error-code coverage, see section 2.1.2). Therefore, test-case generation techniques, that are unable to make assertion about reachability (e.g., fuzzing, plain random, etc.) are unsuited for these kinds of coverage criteria.

3.6 Conclusion and Future Work

Conclusion. In this chapter, we introduced different techniques (i.e., partitioning and hybrid model checking) for optimization of test-case generation. Furthermore, we developed and evaluated different strategies for partitioning and different configurations of hybrid model checking. Furthermore, we evaluated these techniques on a selection of C programs and showed the impact of different adjustable configuration options. Additionally, we provided strategies, that proved to be more effective and efficient than the baseline strategies and, therefore, optimized test-case generation.

Future Work. In future work, we would like to develop and evaluate further partitioning strategies. Especially for partitioning strategies using global information, there are multiple different strategies (e.g., based on data-flow analysis), which might further improve test-case generation. Additionally, we would like to utilize more analysis techniques for hybrid test-case generation, as well as incorporate different test-case generation techniques (e.g., fuzzing), which proved to be useful in different works [24, 25]. Additionally, we would like to measure the progression of effectiveness over time for the predicate and the explicit value analysis to optimize the partial time limits. Next, we would like to evaluate on-the-fly partitioning (i.e., to re-partition after each reachability analysis). Furthermore, we would like to use machine learning approaches to select partitioning strategies based on the program under test (based on static properties of the program, such as the number of loops, unbounded loops, etc.). This is promising, as our results show large differences on partitioning strategies based on single subject systems. Lastly, the partitioning strategies could be used for multi-core or distributed test-case generation. As the partitions can be handled separately, each partition could be run on a different machine and, therefore, improve wall time of test-case generation significantly. The number of available cores could also be used as the number of partitions in this case, to further maximize efficiency in terms of wall time.

4

REGRESSION TESTING

In the previous section, we were concerned with test-case generation for single program versions. However, nowadays, programs evolve at an increasing pace, mostly due to the advent of continuous integration [161]. The goal of regression testing is to validate these changes and, to this end, reveal bugs introduced by erroneous modifications (e.g., incorrect bug fixes, erroneous additional features, etc.) [181].

In literature, regression testing is often used as an umbrella term for test-suite reduction, test-case selection, and test-case prioritization [181]. However, test-case generation tailored for creating test cases in a regression-testing scenario has also been subject to research [169].

In this chapter, we will study the combination of test-suite reduction with test-case generation tailored for regression testing. To this end, we will introduce methods developed in this thesis to further increase the effectiveness of regression-test-case generation. The following research challenges will be tackled in this chapter.

- How to increase effectiveness of test suites tailored for regression testing in terms of actual bug detection ratio (**C1**).
- How to increase efficiency of regression testing...

  ...in terms of CPU time needed to generate test suites tailored for regression testing (**C2.1**).

  ...in terms of the number of test cases of test suites tailored for regression testing (**C2.2**).

In this chapter, we will first extend our running example to further motivate and illustrate different concepts. Next, we will first give an overview of existing state-of-the-art techniques in regression testing and additional fundamentals of regression testing.

In Sect. 4.3.2 we will present a novel concept, developed in the course of this thesis, to create multiple test cases for each test goal to further improve effectiveness of regression testing in terms of error coverage (i.e., the number of bugs revealed).

Section 4.4 shows the overview of our history-based regression testing methodology developed during this thesis. Next, we will show the implementation and the evaluation of our methodology. Lastly, we will discuss related work and give a short summary and show possible future work.

The content of this chapter is based on the following publication:

[154] Sebastian Ruland and Malte Lochau. On the Interaction between Test-Suite Reduction and Regression-Test Selection Strategies. arXiv, 2022. doi: https://doi.org/10.48550/arXiv.2207.12733.

4.1 Running Example

As mentioned in Sect. 2.1.3, the running example in Fig. 2.3 still contains three bugs. First (*Bug 1*), the first element of the array is supposed to be the meta-information about the length of the array. Therefore, the starting index 0 of the **`for-`**loop in line 6 is incorrect.

Second (*Bug 2*), the whole array (other than the first element) should be searched. Therefore, the condition of the **`for-`**loop in line 6 incorrectly terminates the loop at the second last element instead of the last one.

Third (*Bug 3*), the function is supposed to return the index of the last element with an equal value to y. However, due to the incorrect condition of the **`if-`**statement in line 7, the last element with a value equal or less to y is returned.

Next, let us assume a developer validates the initial program with a test suite T_1 consisting of the following test cases.

- $t_1 = \{(x = [1], y = 0), -1\}$,
- $t_2 = \{(x = [4, 6, 6, 4], y = 5), -2\}$.

These test cases fulfill branch coverage and can be created using the methodology explained in Sect. 2.3. When executing these tests for our running example, test case t_2 will fail due to the incorrect return value of 0. Now, the presence of at least one bug is apparent. Therefore, the developer will search the code and fix a bug. In case of Bug 1 as the first bugfix, the developer fixes the program as shown in Fig. 4.1a (where we use standard patch syntax, i.e., $--$ for a line that gets removed and $++$ for a new line, that is added to the program). This leads to the next program version P_1 (see Fig.4.1b).

When executing these tests on our program version P_1, the tests will succeed, and the developer will not discover the remaining bugs. Although test case t_2 is able to reach the error-location of both remaining bugs, it is not able to detect them. The reason is that, as explained in Sect. 2.2.4, to reveal a bug, the test has to reach the location, infect the program state, propagate the infection and, lastly, reveal the propagated infection. In this case, test case t_2 does reach Bug 2 and Bug 3. However, due to Bug 2 Bug 3 will not be able to infect the program state, as the last element is not included in the search. While Bug 2 will infect the program state (as the loop is not iterated often enough) the infected state is not observable, as the result will be correct. In contrast, a test suite T_2 with the following test cases also fulfills branch coverage.

- $t_1 = \{(x = [1], y = 0), -1\}$,
- $t_3 = \{(x = [3, 3, 3], y = 4), -2\}$,
- $t_4 = \{(x = [3, 2, 2], y = 0), -2\}$.

This test suite is able to detect Bug 1 in the initial program P_0, as well as the second bug Bug 2 in program version P_1. Therefore, if test suite T_2 had been available to the developer, the second bug could be fixed as well. This leads to program version P_2, shown in Fig. 4.1d, after applying the patch shown in Fig. 4.1c. The second test suite is now unable to detect the last remaining bug. However, in program version P_2, the first test suite is able to detect the last bug, as the accidental program state fix due to the interaction of both bugs is gone. This shows that both test suites detect the same amount of bugs in the program. However, the effectiveness (i.e., number of bugs found) highly depends on the current program version.

Lastly, we have seen that test case t_2 detects Bug 1 and Bug 3, and test case t_3 detects Bug 2. Therefore, choosing a test suite with both test cases would detect all bugs in our running example. For example, the following test suite T_3 both fulfills branch coverage and detects all bugs.

- $t_1 = \{(x = [1], y = 0), -1\}$,
- $t_2 = \{(x = [4, 6, 6, 4], y = 5), -2\}$,
- $t_3 = \{(x = [3, 3, 3], y = 4), -2\}$.

4.2 Test-Suite Reduction Techniques

As explained in section 2.4, due to the increasing pace of program modification and the accompanying validation, test cases are executed extremely often. In fact, validation is needed so often that test suites cannot be executed fully, as this would take too much time [181]. One prominent method for this problem is test-suite reduction [181]. In this section, we will provide a short overview of different techniques used for test-suite reduction (also called test-suite minimization).

Test-Suite Reduction. Test-suite reduction is concerned with reducing the number of test cases permanently [181]. The test-suite reduction is always accompanied by a criterion (e.g., branch coverage), on which the reduction is based. Usually, the coverage of the given criterion is still the same after reduction. In this case, only *obsolete* test cases (i.e., test cases that do not add coverage, as other test cases already cover everything the obsolete test case would cover) are removed. Test cases can become obsolete during development of the program due to modifications (e.g., removing the code the test case is responsible for testing).

However, the problem to find a minimal set of test cases fulfilling a given criterion is NP-hard, as it is reducible to the minimum set-cover problem [109]. There exist different strategies for test-suite reduction. The optimal solution can be computed utilizing methods from integer-linear programming (ILP). However, to compute a minimal set of test cases, all $2^{|T|}$ subsets of the test suite T need to be enumerated, which is infeasible in case of a large number of test cases.

Different heuristics also exist to increase efficiency of test-suite reduction, for example, by using greedy algorithms [29] or algorithms from big data analysis [53].

```
6--: for (int i=0; i <= x[0]-2; i++)
6++: for (int i=1; i <= x[0]-2; i++)
```

(a) First Bug Fix

```
1  int find_last (int x[], int y) {
2    if(x[0] <= 1)
3      return -1;
4
5    int last = -2;
6    for (int i=1; i <= x[0]-2; i++)
7      if (x[i] <= y)
8        last = i;
9    return last;
10 }
```

(b) New Program Version P_1 After Applying the First Bug Fix

```
6--: for (int i=1; i <= x[0]-2; i++)
6++: for (int i=1; i <= x[0]-1; i++)
```

(c) Second Bug Fix

```
1  int find_last (int x[], int y) {
2    if(x[0] <= 1)
3      return -1;
4
5    int last = -2;
6    for (int i=1; i <= x[0]-1; i++)
7      if (x[i] <= y)
8        last = i;
9    return last;
10 }
```

(d) New Program Version P_2 After Applying the Second Bug Fix

Figure 4.1: Program Versions P_0, P_1 and P_2 with their corresponding Bug Fixes [154]

```
7--: if (x[i] <= y)
7++: if (x[i] == y)
```

(a) Third Bug Fix

```
1  int find_last (int x[], int y) {
2    if(x[0] <= 1)
3      return -1;
4
5    int last = -2;
6    for (int i=1; i <= x[0]-1; i++)
7      if (x[i] == y)
8        last = i;
9    return last;
10 }
```

(b) New Program Version P_3 After Applying the Third Bug Fix

Figure 4.2: Program Version P_3 with the corresponding Bug Fix [154]

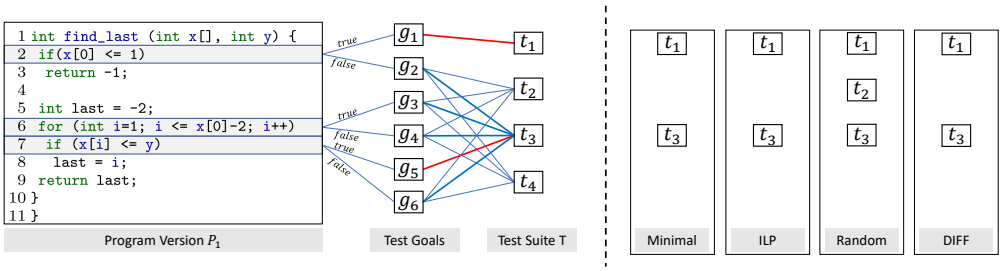


Figure 4.3: Comparison of Test-Suite Reduction Strategies (adapted from [154])

In the following, we will give a short illustration of these techniques, as well as their strengths and weaknesses.

Figure 4.3 (adapted from [154]) illustrates the test-suite reduction problem. The program version P_1 contains six branches (i.e., the **`true-`** and **`false-`** branches of the lines 2, 6 and 7). Therefore, to branch coverage constitutes six different test goals ($G = \{g_1, g_2, \ldots, g_6\}$). To cover all test goals, the following test suite can be used.

- $t_1 = \{(x = [1], y = 0), -\}$
- $t_2 = \{(x = [4,6,6,4], y = 5), -\}$
- $t_3 = \{(x = [3,3,3], y = 4), -\}$
- $t_4 = \{(x = [3,2,2], y = 0), -\}$

However, the test suite is not minimal, as t_2 and t_4 cover the same test goals. Additionally, all test goals covered by t_2 and t_4 are also covered by t_3, while t_3 also covers the test goal g_5. In fact, both test cases t_1 and t_3 are indispensable to fulfill branch coverage. Therefore, the minimal test suite for branch coverage on the program P_1 consists of only the test cases t_1 and t_3, as shown on the left in Fig. 4.3.

ILP-based Test-Suite Reduction. A test-suite reduction problem can also be encoded as an Integer linear optimization problem. This encoding can then be solved by using Integer Linear Programming (ILP) solvers [109]. An exemplary encoding for our running example is to introduce a decision variable x_i for each test case t_i in the test suite T. The decision variable is either equal to 1 if t_i is selected or 0 otherwise. The ILP formula contains for each test goal a single clause. The clause builds the sum of all decision variables for each test case, covering the test goal. The clause also requires the sum to be greater than 0 and, therefore, requires at least one test case to be selected that covers the test goal. Since the solution to the ILP formula requires each clause to hold, all test goals will be covered by the resulting test suite. Lastly, we can enforce minimality of the resulting test suite by specifying the overall optimization goal to minimize the sum of the values of the decision variables x_i.

When applying the encoding to our running example, we will end up with the decision variable x_1 (t_1), x_2 (t_2), x_3 (t_3) and x_4 (t_4). The clauses for the ILP solver would be as follows.

$$\begin{aligned} x_1 &>= 1 \\ x_2 + x_3 + x_4 &>= 1 \\ x_2 + x_3 + x_4 &>= 1 \\ x_2 + x_3 + x_4 &>= 1 \\ x_3 &>= 1 \\ x_2 + x_3 + x_4 &>= 1 \end{aligned}$$

The optimization objectives is defined as:

$$min(x_1 + x_2 + x_3 + x_4).$$

Therefore, ILP solvers are able to provide an optimal solution for the test-suite reduction. While recent ILP solvers are able to use different heuristics to decrease the CPU time needed for computation, they still lead to high computational effort in the worst-case scenario.

Greedy-based Test-Suite Reduction. Greedy algorithms usually start with an empty test suite and incrementally select test cases until all test goals are covered [29, 181], or start with a full test suite and remove redundant test cases until no test case can be removed anymore. In the first case, the algorithm selects during each step the test case covering the most uncovered test goals. Applied to our running example, the first test case selected would be t_3, as it covers 5 uncovered goals. Next, the test cases t_2 and t_4 do not cover additional test goals, however, test case t_1 covers one additional uncovered goal. The resulting test suite would (coincidentally) also be the minimal test suite.

- t_3 (covering 5 uncovered test goals),
- t_1 (covering 1 uncovered test goal).

This is due to the fact that greedy algorithms will run into a local optimum. In case of our running example, the local optimum is also the global optimum. However, if there exist other local optima, there is no guarantee about the resulting test suite size.

Random-based Test-Suite Reduction The same as greedy-based test-suite reduction, random-based test-suite reduction either starts with an empty test suite and adds test cases until all test goals are covered, or starts with the full test suite and removes test-cases until no test case can be removed anymore without reducing the number of covered goals. In this thesis, we start with an empty test-suite. Afterwards, a random test case is selected and added to the test suite if it increases the number of covered goals. Otherwise, the test case will be discarded. This step is repeated until no further test cases can be selected or the

number of covered goals of the reduced test suite is the same as the original test suite.

Applied to our running example, the first test case selected could be t_2. Next, if test case t_4 is selected, the test case will be discarded, as it covers the same goals as t_2. In the next step either t_1 or t_4 will be selected and in the last step the remaining test case of both. The resulting test suite would consist of the test cases t_2, t_1 and t_3. As this approach is purely random-based, the reduced test suite can be anything from the original test suite (as long as there are not multiple test-cases covering exactly the same goals) to the minimal test suite.

This technique is, on average, much more efficient than ILP, and in general, also more efficient than a greedy-based technique. However, it might lead to larger test suites as random-based test-suite reduction is purely random.

As shown, the different techniques might lead to different results concerning the number of test cases in the reduced test suite and CPU time. However, removing test cases might also reduce effectiveness of the resulting test suite. There are two main reasons for a reduction of effectiveness. First, the coverage criterion used for test-suite reduction might not correspond to testing effectiveness (i.e., to reduce based on branch coverage might not correspond to actual bug-finding probability). Second, a test case might be redundant in the current version. However, it might become relevant later on. In our example, test case t_2 does not add coverage to the reduced test suites and also does not detect any bug in the current version. However, after fixing the next bug in the program version P_1 in our running example, test case t_2 will again detect the last remaining bug *Bug 3*, while the other test cases will not detect Bug 3. Therefore, even though test case t_2 is irrelevant for the current version, reducing it will reduce effectiveness of the test suite later on. For this reason, random-based test-suite reduction in our example will lead to the most effective test-suite for our running example, even though it provides the least optimal test-suite reduction of the techniques shown in this section.

4.3 Regression-Test-Case Generation

In the previous chapter, we were concerned with test-case generation for single program versions. In this section, however, we will generate test cases tailored for testing subsequent program versions. Figure 4.4 shows the overview of the methodology in this section.

First, an initial version is patched to a subsequent version. Next, test-case generation can be used, which uses information about the changes. In this section, we will provide two main techniques to increase the effectiveness of such test cases. First, by generating so-called revealing test cases. Second, by generating multiple test cases for each test goal. Both techniques increase effectiveness of the test suite, usually at the cost of efficiency. In Sect. 4.6, we will evaluate these techniques and calculate the trade-off between effectiveness and efficiency. In the remainder of this section, we will explain the techniques in more detail.

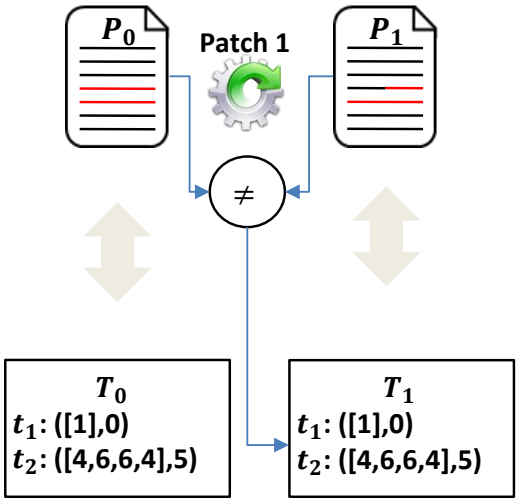


Figure 4.4: Test-Case Generation for Regression Testing

4.3.1 *Change-Aware Test-Case Generation*

For regression testing, either an existing test-suite can be used, or a new test-suite (or single test cases) can be generated. In both cases, the criterion the test-suite has to fulfill can be the same. As explained in chapter 2 branch coverage or similar coverage criteria are often used. For regression testing, reaching the modified parts of the code (i.e., traversing test-cases) is often considered as coverage criterion [181]. The test case has first to reach the program location to detect a bug introduced by faulty program modifications (or existing bugs, which became detectable to program modifications).

However, reaching the program location is insufficient. The test case also has to provide input values that will alter the program state (compared to the previous program version), propagate the altered program state, and finally reveal the altered program state [11]. While this might happen for traversing test cases, it is not guaranteed. Revealing test cases (also called differential test cases in literature [169]), however, guarantee to show different output behavior of the modified program version compared to the original program version. For this reason, revealing test cases are more likely to detect bugs [169].

To create traversing- or revealing test cases, the program can be modified such that the test goal can be encoded as reachability problem.

Traversing Test-Case Generation. Test goals corresponding to traversing test cases are in general already encoded as reachability problem (as traversing test cases need to reach the modification). However, we still need to specify, which parts of the code have been changed.

To this end, we can use any diff tool to create a patch from the old program version to the new one. For every changed line of code (i.e., removed, added or modified), we add a special label to the source code. For our running example from program version P_0 to P_1, only the initialization of the iterator variable of the for-loop in line 6 is modified. Therefore, we add a new label above line 6. The

```
int find_last_p0_1(int x[], int y) {
  if(x[0] <= 1)
    return -1;

  int last = -2;
  change_1:
  for (int i=1; i <= x[0]-2; i++)
    if (x[i] <= y)
      last = i;
  return last;
}
```

Figure 4.5: Comparator-Function for the Running Example

```
void find_last_p0_1(int x[], int y){
  if(find_last_p0(x,y) != find_last_p1(x,y))
    test_goal:printf("differencing_test_case_found");
}
```

Figure 4.6: Comparator-Function for the Running Example

resulting function is shown in Fig. 4.5. When executing a test-case generator, we can specify each label starting with "change_" as test goal and, therefore, generate test cases for each modified line.

Revealing Test-Case Generation. It is also possible to encode the generation of revealing test cases as a reachability problem. In this case, a off-the-shelf test-case generation tool can be used to generate the revealing test case. For this purpose, the functions and global variables of the program versions need to be renamed (otherwise, there will be naming conflicts). Next, a comparator-function needs to be generated. For our running example, a comparator-function for the program versions P_0 and P_1 is shown in Fig. 4.6. The comparator-function in Fig. 4.6 assumes that the array x is not modified by **`find_last_p0`** and **`find_last_p1`**. Otherwise, the array would need to be copied and passed as argument. However, copying relies on the size of the array, which is tricky to evaluate automatically for C programs. Therefore (and for other reasons), the generation of corresponding comparator-functions is only for specific types of programs under test fully automatable. For example, for arrays, either an assumption about the usage of the array or the size of the array have to be made (or in a semi-automated fashion, the user needs to be asked).

Additionally, not all modifications can be used to create a comparator-program. For example, if the signature of the method changes (e.g., the return type of the function changes), comparator-functions cannot be created, at least in an automated manner. This problem is shown in Fig. 4.7. In this case, a fifth program version is created by changing the return type of the function to a structure called **`intStruct`**. When creating a comparator-function, as shown before, the resulting program will not be compilable. However, despite the technical restrictions of such comparator-functions, they still provide means to generate revealing test cases in an automated fashion (at least for most code modifications).

```
int find_last_p3 (int x[], int y) {...}
struct intStruct find_last_p4 (int x[], int y) {...}


void find_last_p3_4(int x[], int y){
  if(find_last_p3(x,y) != find_last_p4(x,y))
    test_goal:printf("differencing_test_case_found");
}
```

Figure 4.7: Invalid Comparator Program

4.3.2 *Multi-Test-Case Generation*

When generating test cases for a modified program version, creating a single test case per test goal (i.e., modification-traversing or modification-revealing) the test case naturally does not guarantee any bug detection. To further increase the bug detection probability (i.e., effectiveness), multiple test cases per test goal can be used. For example, both test-cases t_2 ($t_2 = \{(x = [4,6,6,4], y = 5), -\}$) and t_3 ($t_3 = \{(x = [3,3,3], y = 4), -\}$) of our running example reach the bugged line 6 in program version P_1. However, only t_3 is able to detect the bug. If a single test case is generated, t_2 would be sufficient. However, the bug would remain undetected. By increasing the number of test cases, the odds to generate a test case that detects the bug (e.g., t_3) increase. Therefore, we developed a novel technique to generate multiple test cases (guaranteed to traverse different paths when executed) for each test goal.

To this end, we utilize the technique on-the-fly weaving (see Sect. 3.1). However, compared to the previous weaving, we create a so-called negated path when reaching a test goal. For the test goal, a test case is created based on the counterexample as before. Afterward, all branches (also called assumption edges) present in the counterexample are collected and added to the negated path in the same order as in the counterexample. Therefore, the negated path is a partial path (consisting only of branches). The next reachability analysis for the test goal will be altered so that the negated path is not available as a path for the test goal anymore. Note that as before, the weaving is done to modify the reachability analysis. The program itself is not modified. Since the reachability analysis unrolls loops, we can differentiate between different loop iterations.

To illustrate this concept, we will explain the method on our running example (see Fig. 2.3 in Sect. 2.1.3). As test goal, we use the statement *return last*; in line 9. First, we execute the reachability analysis, which builds the ARG shown in Fig. 4.8. As before, the first part of the abstract state corresponds to the location state and the second part corresponds to the predicate state. The third part corresponds to the index of the negated path. Upon fulfilling the test goal by reaching state $0'$, an counter example is extracted (highlighted as red path in Fig. 4.8).

All branches (i.e., (3) $\xrightarrow{!(x[0]<=1)}$ (5) and (7) $\xrightarrow{!(i<=x[0]-2)}$ (8) in our example) are extracted and used to create a new negated path from the counterexample, which is added to the test goal. When executing the next reachability analysis (see Fig. 4.9), a new variable (variable np in state $1'$) is weaved into the ARG for the negated

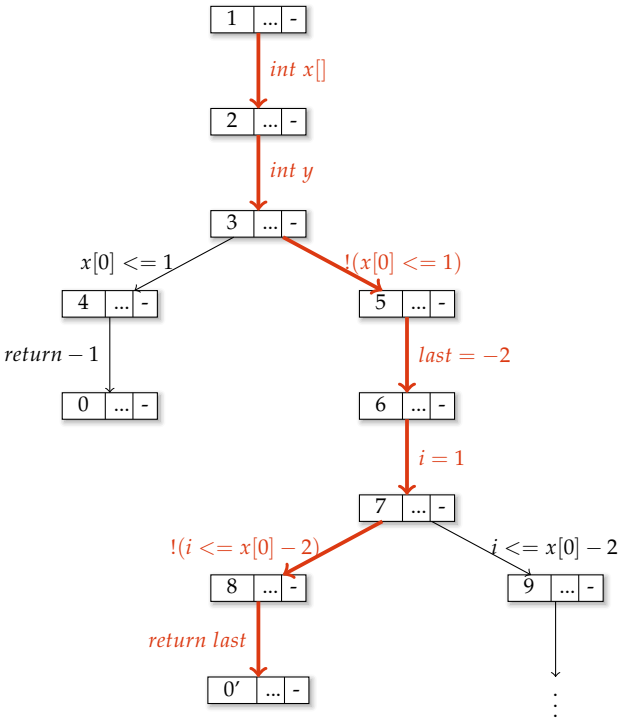


Figure 4.8: First Reachability Analysis

path. Additionally, each ARG state (more specifically the abstract state of the negated path) carries an index of the current edge of the negated path. The index will always be the same value as the weaved variable. The reason we need the additionally weaved variable is for constructing the counterexample as soon as a test goal has been reached.

Now, whenever we traverse a branching edge, we check whether the branching edge is the next edge of the negated path. Therefore, from state 1 to state 3 the next edge of our negated path is (3) $\xrightarrow{!(x[0]<=1)}$ (5). Since the edge from state 3 to state 4 is a different edge, we weave an assignment of the variable of the negated path to -1 and set the index of the negated path to -1 (i.e., marking the current path as available for the test goal), as well . As soon as the index is -1 we won't need further weaving (since we traverse a different path than the negated path).

When reaching a branching edge that is the next edge of the negated path (e.g., the edge from state 3 to 5), we will weave an increment of the variable and also increment the index of the negated path. Therefore, the next edge of our negated path from state 5 to 7 will be (7) $\xrightarrow{!(i<=x[0]-2)}$ (8). When reaching this edge we will again increment the variable and the index (see state $8'$). When traversing a different branching edge, we will set the value of the variable and the index to -1 (see state $9'$).

When reaching the test goal, a new condition is weaved in the ARG. The condition checks whether the value of the weaved variable is equal to the number of branch edges in the negated path (in this case, the same path as the negated path has been taken). If the condition is true, the test goal is not reached (see state $0''$). However, the reachability analysis may continue (this prevents accidental pruning

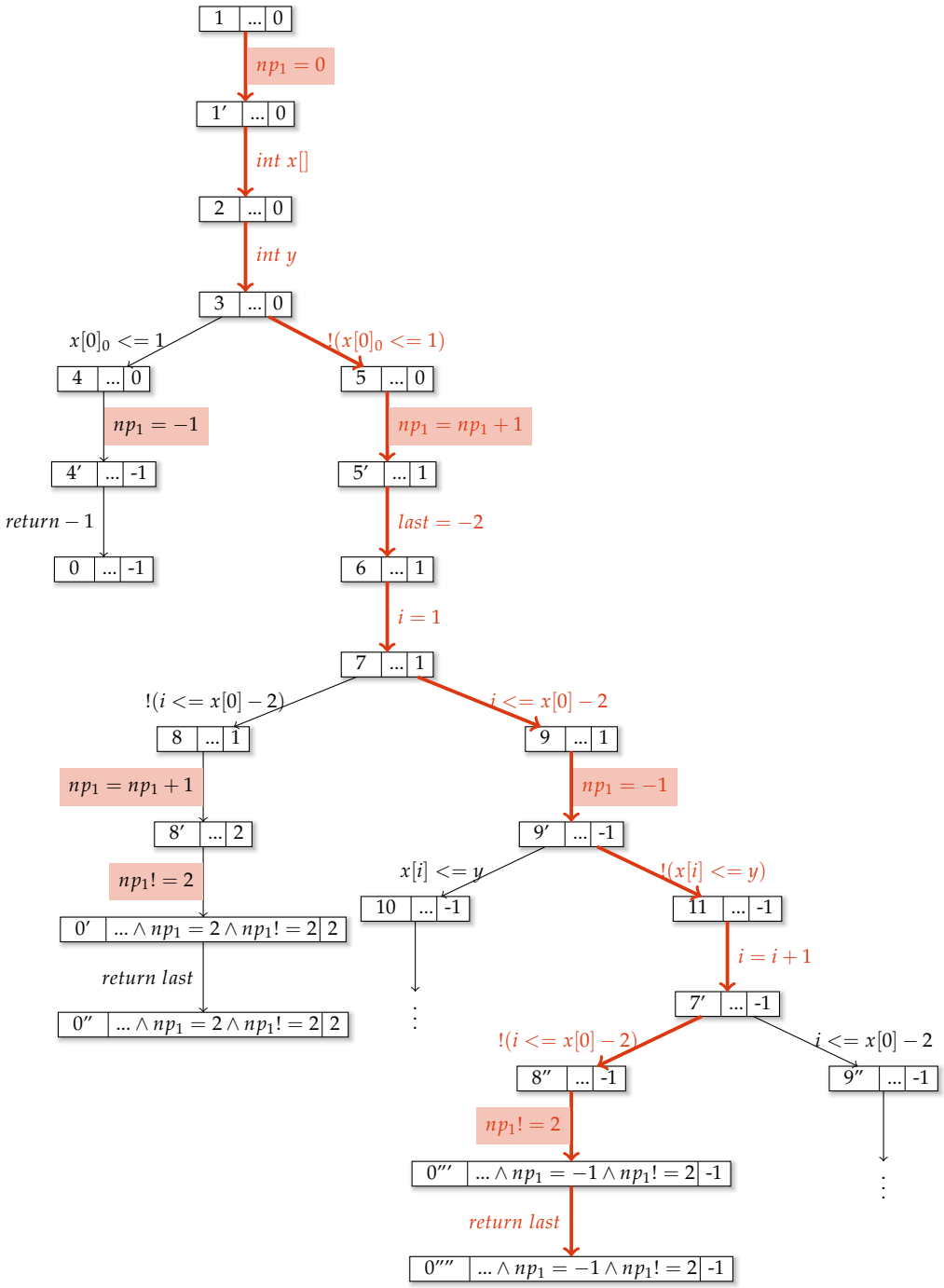


Figure 4.9: Second Reachability Analysis

of the ARG). Otherwise, the test goal has been reached by traversing a different path (see state $0''''$). In this case, a counterexample for state $0''''$ can be computed, and a test case can be extracted again. This step can be repeated by adding new negated paths until no further paths are available to reach the test goal. Since a path is sensitive to the number of loop iterations, we could potentially create unlimited test cases for our running example (or at least until a integer overflow for the loop index happens).

Since loops are unrolled during ARG creation, this methodology is able to differentiate between different numbers of loop iterations, as well as nested loops. However, in case of source code optimizations done during either CFA construction of ARG construction (e.g., removing loops due to loop invariant code motion, pruning loops due to induction etc. [5]) the number of possible test cases might differ from the expected amount. Still, in case multiple test cases are created they will differ in the paths of the counterexamples. Lastly, in case of non-determinism present in the source code (e.g., due external function calls, system calls, etc.) two test cases that should have taken different paths might, when executed on a real system, still take the same path.

4.4 History-based Test-Case Generation

In the previous section, we provided a technique to further increase the effectiveness of generated test-suites based on a single version update (i.e., a commit).

However, nowadays, commits become more and more frequent during development. Therefore, projects are usually accompanied by large version histories. To this end, we introduce a novel technique to use several prior program versions for test-case generation. For this purpose, comparator-programs (see Sect. 4.3.1) can be written for different versions (e.g., for program version P_0 compared to P_1). However, when testing full version histories, different interactions between different parameters/techniques might happen. Therefore, we provide means of configuring different parameters for regression testing to evaluate and tune effectiveness and efficiency during regression testing. Figure 4.10 shows the overview of our approach. The number of the parameters in the figure correspond to the number of the explanation of the parameters in the following.

① **Regression Test-Case Selection Criterion (RTC):** New regression test cases for a program version P_i may be added to test suite T_i either by means of modification-traversing test cases (i.e., at least reaching the lines of code modified from P_{i-1} to P_i, see Sect. 4.3.1) or by modification-revealing test cases (i.e., yielding different outputs if applied to P_i and P_{i-1}, see Sect. 4.3.1).

② **Number of Regression Test Cases (NRT):** The number of *different* regression test cases added into T_i satisfying **RTC** for *each* previous program version T_j, $0 \leq j < i$, according to parameter **NPR** (see Sect.4.3.2).

③ **Number of Previous Program Revisions (NPR):** The (maximum) number of previous program versions P_j ($i - \textbf{NPR} \leq j < i$) for all of which **NRT** different test cases satisfying **RTC** are added to T_i.

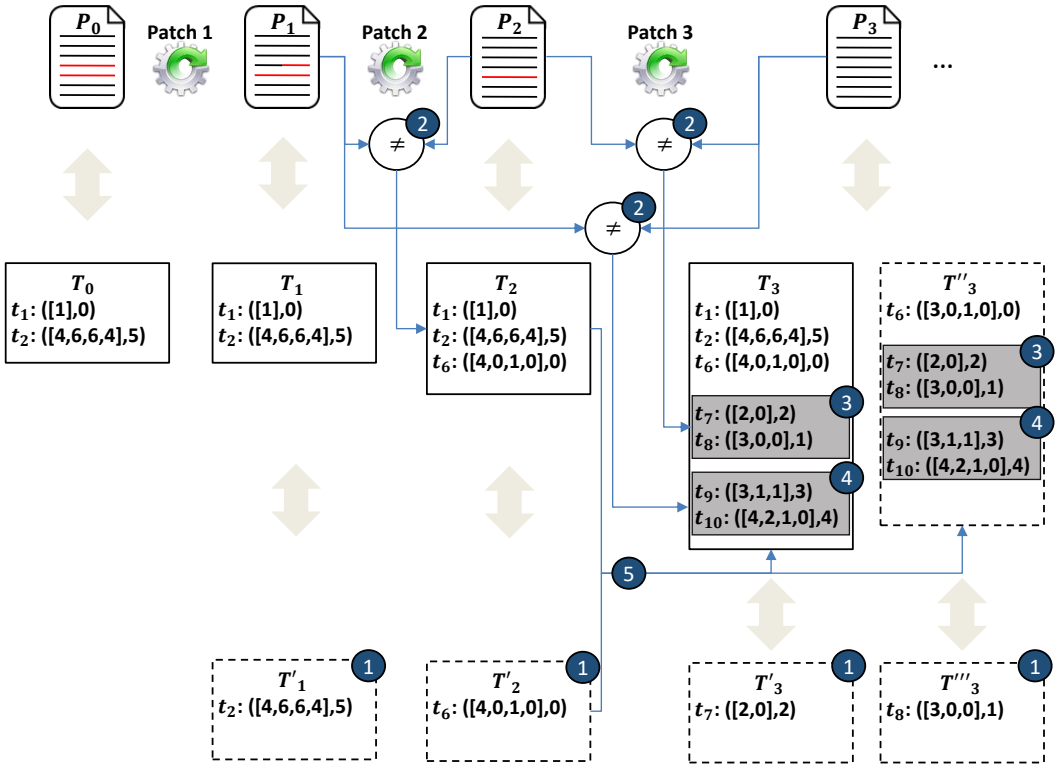


Figure 4.10: Big Picture for History-based Regression Testing(adapted from [154])

④ **Reduction Strategy (RS):** Technique used for test-suite reduction. Possible strategies: **None**, **ILP**, **Random**, and **DIFF** (see Sect. 4.2).

⑤ **Continuous Reduction (CR)):** Controls whether the non-reduced reduced test suite T_{i-1} (**No-CR**) or the reduced test suite T'_{i-1} (**CR**) of the previous program P_{i-1} version is (re-)used for the next program version P_i or if the previous test cases are ignored (**None**). The selected test suite is than extended by new test cases according to the previously described parameters.

Therefore, we can use different configurations of the most relevant techniques for regression testing and their interactions to further optimize regression testing.

4.5 Implementation

We implemented comparator-generator to create programs with comparator-functions, as explained in Sect. 4.3 and a case-study crawler to collect subject systems. Additionally, to evaluate the illustrated techniques, we implemented the test-case generation techniques in the CPAchecker framework and extended the test-suite validator TestCov for the test-suite reduction techniques in the course of this thesis. Additionally, we implemented a tool for generator comparator-functions as shown in Sect. 4.3. We called the framework incorporating these tools RegreTS (**Regre**ssion **T**esting **S**trategies).

```
1  int compareIntPointer(const int const* v1, const int const* v2){
2    if ((v1 == 0 && v2 != 0) || (v1 != 0 && v2 == 0)){
3          return 0;
4      }else
5          if (v1 != 0 && v2 != 0){
6              int v1Int = *(v1);
7              int v2Int = *(v2);
8              if (v1Int != v2Int){
9                  return 0;
10             }
11         }
12     return 1;
13 }
```

Figure 4.11: Exemplary comparator-function for int Pointer

4.5.1 *Comparator-Generator*

To create comparator-functions (see Sect. 4.3.1) we first parse both program versions using Eclipse CDT [1] to an abstract syntax tree (AST). For more information about ASTs we refer to the work of Harper [85] or Aho et al. [5]. Additionally, we use CPAchecker to parse the program to a CFA, which contains additional information.

We use the CFA to get information about the types of the function parameters, the return value and the global variables being used. The AST is used for source code manipulation (e.g., removing functions, renaming variables, adding functions, etc.).

There are three types currently supported from our framework, namely basic type (e.g., integer, double, etc.), composite types (structs and unions) and basic or composite type pointers. Additionally, we resolve typedefs as far as possible. In case the type for which a comparator is needed is a basic type, we inline an unequal check as comparison (e.g., i_1 `!=` i_2, cf. 4.6). If the type is a pointer, we create a new function that takes two parameters with the same type as the pointer. First, the function checks whether exactly one pointer is null. In this case, the function returns 0. Otherwise, if both pointers are not null, we dereference the pointers and create a comparator-function for the dereferenced type. In case the dereferenced values differ, the function returns 0, as well. If both dereferenced values are equal or both pointers are null, the function returns 1. An exemplary comparator-function for pointers is shown in Fig. 4.11.

In case the type is a composite type, we create a new function that takes two parameters with the same type as the composite type. Next, we generate a comparator-function for each member of the composite type. In case any the comparator-function for any member returns false, the comparator-function of the composite type returns false as well. An exemplary comparator-function for structs is shown in Fig. 4.11.

Currently, we do not support arrays. This is due to the fact, that in C there is usually no information about the array length. There exist different notions

[1] https://www.eclipse.org/cdt/

```
1  struct str1 {
2    int m1;
3    int* m2;
4    struct str2 m3;
5  };
6  ...
7  int compareStructstr1( struct str1 v1, struct str1 v2){
8    if(v1.m1 != v2.m1){
9      return 0;
10   }
11   if(!compareIntPointer(v1.m2, v2.m2)){
12     return 0;
13   }
14   if(!compareStructstr2(v1.m3, v2.m3)){
15     return 0;
16   }
17   return 1;
18 }
```

Figure 4.12: Exemplary Comparator-Function for Structs

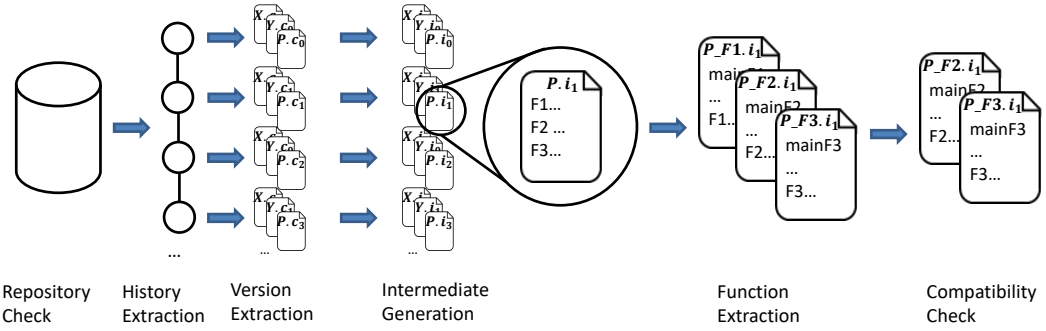


Figure 4.13: Overview of Case Study Search

(e.g., having a terminating character, having the size as first element, etc.) to track the array size, which usually cannot be reconstructed automatically. Therefore, comparing each element of two arrays is difficult to do automatically. Additionally, Strings are often encoded as char pointers (i.e., a pointer to the first element of an char array). This is currently unsupported and only the first character would be compared.

4.5.2 *Case-Study Crawler*

To gain more subject systems for our evaluation, we implemented a case-study crawler. The workflow of the crawler is shown in Fig 4.13. First, we use the GitHub API to search for popular public C repositories (in the current configuration of the crawler at least 200 stars). Next, we clone the repository using git and extract information from the git history which changes affect which C files. During the version extraction, we checkout a single commit (starting from the latest commit). We then try to build the project by automizing common build techniques (e.g., make, autoconf, autogen, etc.). Additionally, we modified the compiler calls

to generate intermediate files whenever executed, as those intermediate files are used as subject systems. If the build did not work, we try to generate intermediate files by calling the compiler for each c files, including necessary headers from the project. If those steps fail, the repository gets marked for manual inspection and removed from list of potential subject systems for now. If the build succeeds, the intermediate files are copied and the version extraction and intermediate generation is repeated for the remaining commits.

After generating intermediate files for each commit, we check for each function in each intermediate file the number of changes during the version history. A change can either be a change directly within the function body or a change of a function that gets called. In both cases the behavior of the function might change during execution. Since the generation of comparator-programs (see Sect. 4.3.1 only works for function with the same parameters, we use the function name, as well as the function definition to identify a function. Therefore, if a function with the same name but different parameters or return value exists within the version history we will consider those as different functions.

For each function under consideration, we create a new intermediate file (see Function Extraction in Fig. 4.13) consisting of the function itself and all functions called directly or indirectly, as well as all needed global variables, typedefs, etc. Unnecessary source code is removed. Additionally, we generate a new main function consisting of initialization of global variables and parameters, as well a a function call to the function under test.

Next, we count the number of different versions of the function. In case to few versions exist, we remove the function from our potential subject systems.

Lastly, we check whether the functions can be used as subject systems by generating two comparator-function for the latest version. The first comparator-function executes the function twice with the same parameters (and copied global variables to prevent interaction between both functions) and checks if the return values are the same. For this comparator-function we execute a test-case generation and try to create a test-case resulting in different return values. If such a test-case can be found, the return value of the function does not solely rely on the parameter and the state of the global variables but also other properties (e.g., system clock, file system state, etc.). In this case the function does not satisfy the requirements for our subject systems. The second comparator-function executes the function twice, however, with different parameters and global variables states. For this comparator-function we execute a test-case generation trying to find a test-case which leads to different return values. If no such test case can be found the return value of the function does not rely on the parameters or state of the global variables (e.g., always returning $'0'$). In this case the function does not satisfy the requirements for our subject systems, as well. For both test-case generation steps we set a time limit of 900s CPU time to prevent potential indefinitely test-case generation executions (since we use reachability analyses for test-case generation, which is per definition undecidable).

Each function, that has sufficient versions and passes the compatibility check is added to our subject systems.

4.5.3 *Test-Suite Reduction*

For measuring the coverage of test-cases and test-suites, we use TestCov. TestCov is a tool for robut test-suite execution and coverage measurements of C programs. [29]. Additionally, TestCov has two build-in test-suite reductions techniques, namely *DIFF* and *By_Order*. *By_Order* reduces the test-suite by first starting with an empty test-suite. The test cases are next selected in the order of their filenames. If a test case increases the coverage of the reduced test-suite it is added. Otherwise, the test case gets discarded. *DIFF* reduces the test suite in a greedy manner, as explained in Sect. 4.2.

Each test-suite reduction strategy is implemented as function, receiving a list of test cases paired with the coverage of each test case. Therefore, to add the random- and ILP-based test-suite reduction strategies (see Sect. 4.2), we added a two new functions implementing the strategies.

The random-based test-suite reduction takes a random test case and checks, whether it increases the coverage of the reduced test-suite. If the test-case increases the coverage, it is added to the reduced test-suite. Otherwise, the test case is discarded.

The ILP-based test-suite reduction uses Python MIP (Mixed-Integer Linear Programming) tools, a collection of python tools for solving mixed-integer linear problems. To this end, we encode the test-suite reduction problem as integer linear problem (ILP) as shown in Sect. 4.2. Next, we use MIP to solve the problem and build the reduced test suite depending by adding each test case selected by the ILP solution.

4.5.4 *Test-Case Generation*

To evaluate the illustrated test-case generation techniques, we implemented them in the CPAchecker framework in the course of this thesis as addition to the implementation in Sect. 3.3. Figure 4.14 shows the additional modules. Each module with a thick border is either new or changed. The new and changed modules are explained in the remainder of this section.

Test Goal Test goals now consists of a list of CFA edges and additionally, a set of negated paths. Initially the set of negated paths is empty and negated paths will be added after reching the test goal depending of the configuration of the test-case generation.

Tiger MultiGoal Algorithm. When extracting a test case from the counterexample upon reaching a test goal the Tiger MultiGoal Algorithm now checks whether more test cases for the test goal should be created (depending on the configuration). If more test cases should be created, all assumption edges from the counterexample are extracted and added as negated path to the test goal (see Sect. 4.3). Next, the test goal will be added to the next partition. If no other partitions are present a new partition will be created and the test goal will be added. The test goal with newly negated paths should never be added to the current par-

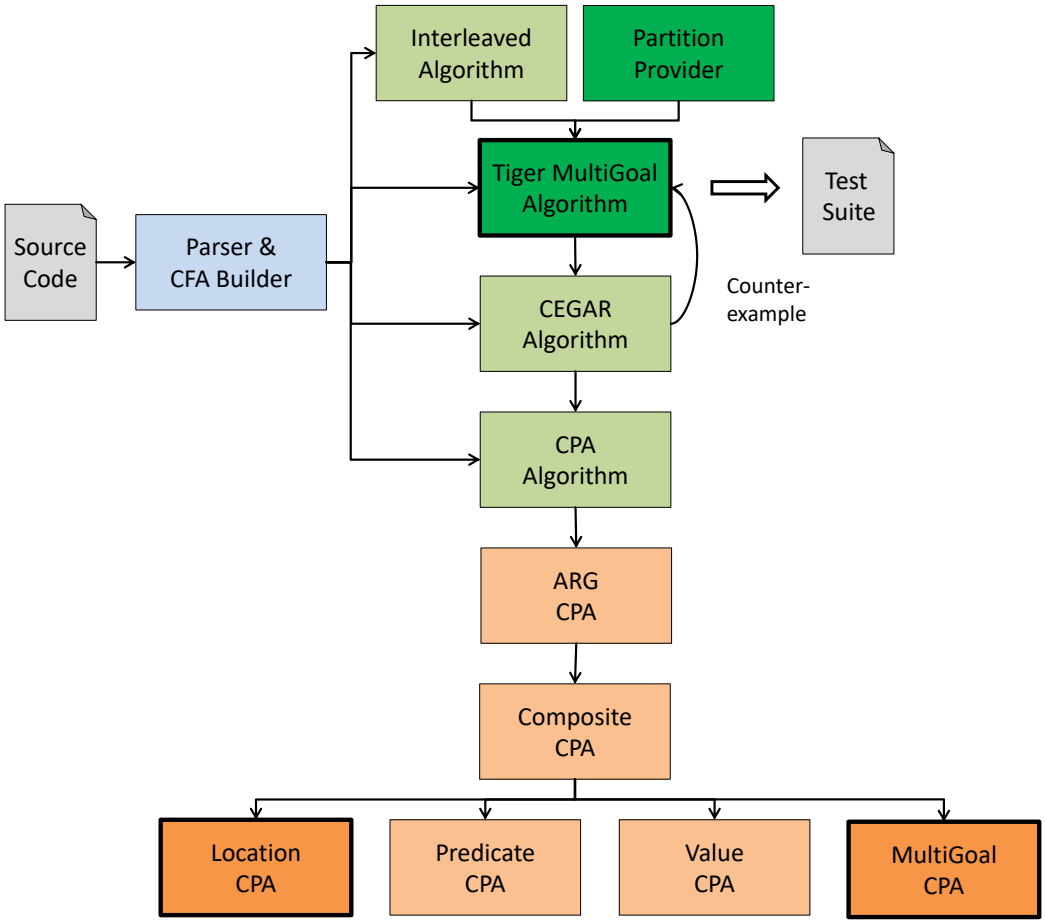


Figure 4.14: Modules for Test-Case Generation with CPAchecker (adapted from [21])

tition. The reason is that for the current partition an ARG has already been created. However, to weave the variable necessary for the negated path a new ARG needs to be constructed. Therefore, if the test goal will be re-added to the current partition, the test goal will be reached again without declaring the variable for the negated goal. However, since the check for the negated path (see Sect. 4.3) will be added anyway, counterexample construction will break due to the usage of non-declared variables.

LocationCPA. The LocationCPA is responsible for the traversing of the CFA during ARG construction. Therefore, weaving of new edges has been added to the LocationCPA directly. However, the LocationCPA is not responsble for the decision what kind of weaving should be done. Therefore, we created a new interface **`WeavingState`** which is responsible for the communication of the LocationCPA and other CPAs that need weaving. The interface has two methods. First, **`getEdgesToWeave`** which tells the LocationCPA which variable and which weaving type (i.e., declaration, increment or assumption) should be weaved in the current location. Second, **`addWeavedEdges`** which the LocationCPA uses to tell the other abstract state which edges have been weaved. The need for this method will be explained in the next paragraph.

The weaving itself is handled by the strengthening-operator of the LocationCPA (see Sect. 2.3.2.1). The strengthening-operator allows the modification of successor states based on the successor states of other CPAs. For weaving, the strengthening-operator of the LocationCPA now checks, whether another state implements the **`WeavingState`** interface. If there exist other abstract states implementing this interface, the strengthening-operator next checks if variables should be weaved and weaving will be executed.

The current abstract state of the LocationCPA will be replaced by a new location state that has the first weaved edge as leaving edge. For every other weaved edge a new location state will be constructed and added as successor (i.e., the state that will be reached when traversing the edge). The last weaved edge will get the original location state as successor. This techniques allows us to weave all variables in order in the reachability analysis (since each weaved edge is the leaving edge of the prior location state).

Additionally, the LocationCPA is responsible for caching of weaved edges. This is essential, since in case of a CEGAR iteration (see Sect. 2.3.2.2) the ARG will be constructed again with additional information. The additional information might be the reachability of a state due to negated paths. In case the the weaving is not cached, the newly created state for the test goal will again be unreachable. However, as the state is not equal to the state in the prior reachability analysis CEGAR will be executed again, resulting in an endless loop of CEGAR iterations. Therefore, caching is essential as this will weave exactly the same state and edges and will ascertain that the information of the previous CEGAR iterations are still valid and usable.

MultiGoalCPA. The state of the MultiGoalCPA now includes the current index of each negated path additional to the index of the test goal. Whenever

computing a successor state for a multigoal state, the transfer relation checks for each negated path whether the CFA edge if the next edge on the negated Path. If it is, the variable needs to be incremented. Otherwise, the variable needs to be set to -1 (if wasn't assigned to -1 already). The index of the negated goals will be assigned the same value as the weaved variable.

Each variable that needs to be weaved will be stored in the new successor state and is available to the LocationCPA through the **`WeavedState`** interface the multigoal state implements.

Additionally, the multigoal state has a set of weaved edges (that are passed to the multigoal state from the LocationCPA after weaving). When reaching a test goal with negated paths, first the assumption for the negated path needs to be weaved in. However, the test goal is already fulfilled due to the last traversed CFA edge and the CFA will not be traversed anymore. For this reason, we need to delay the construction of a counterexample. To this end, the MultiGoalCPA will not mark the current state as target state (and, therefore, initiate counterexample construction) as long as variables need to be weaved or weaved edges have not been traversed. When reaching a weaved edge, a new successor state for the multigoal state will be constructed by copying the predecessor state and removing the weaved edge from the set of weaved edges. As soon as the set is empty (and the test goal has been marked as reached) a counterexample will be constructed.

Lastly, we modified the merging-operator of the MultiGoalCPA. Merging is only enabled for two states if the set of weaved edges is empty for both. When merging two states with different indexes for a negated path, the larger index will be used (the indexes can only be either equal or one is -1 and the other is larger than 0).

4.6 Evaluation

The framework shown in this chapter enables us to investigate the impact of the different parameters ①–⑤ on efficiency and effectiveness during regression testing of version histories.

In our evaluation, we consider version histories, based on git, of C programs. To provide realistic effectiveness and efficiency measures, we use real-world systems, which we obtained from GitHub. As these systems are git-based, they come with a version history, which we utilize to build the subsequent program versions. However, as these systems usually do not come with a bug history, we simulate bugs throughout the version history by utilizing techniques from mutation testing. To simulate the bugs, we repeatably apply different mutation operators for C programs. Next, we use the simulated bugs to measure the bug-detection ratio of our generated test-suites to measure effectiveness. Lastly, we also measure the efficiency of the generated test cases in terms of computational effort for the generation (and reduction) of the test suites, as well as the size of the test suites (i.e., the number of test cases) throughout the version history.

4.6.1 *Research Questions*

We consider the following research questions.

(RQ1) How does the regression-testing strategy impact *testing effectiveness*?

(RQ2) How does the regression-testing strategy impact *testing efficiency*?

- **(RQ2.1)** How does the regression-testing strategy impact $efficiency_{CPU}$?
- **(RQ2.2)** How does the regression-testing strategy impact $efficiency_{size}$?

(RQ3) Which regression-testing strategy constitutes the *best trade-off between effectiveness and efficiency*$_{CPU}$?

4.6.2 *Experimental Setup*

Next, we will describe the evaluation methods and experimental design used in our experiments to address the research questions.

To evaluate different regression-testing strategies, we use the following values for the parameters of our methodology. For parameter **RTC**, we consider either modification-traversing (**MT**) or modification-revealing (**MT**). For the number of test cases per test goal (**NRT**), we limit our consideration of possible values to $\in \{\mathbf{1}, \mathbf{2}, \mathbf{3}\}$, as we noticed a diminishing increase of effectiveness for higher numbers. For the same reason, we limit our consideration of possible values for the number of previous program versions (**NPR**) to $\in \{\mathbf{1}, \mathbf{2}, \mathbf{3}\}$. Additionally, we do not consider 0 for *NRT* and *NPR*, as in both cases, this would obviously lead to an empty test-suite. For parameter **RS**, we allow each of our currently supported test-suite reduction techniques (i.e., **ILP**, **DIFF** or **Random**) or **None** if no test-suite reduction should be used. Lastly, we allow for either continuous test-suite reduction (**CR**), non-continuous test-suite reduction (**No-CR**, i.e., using the non-reduced previous test-suite and reduce once) or ignoring the previous test-suites (**None**) by specifying the parameter **CR**.

Each combination of the possible instantiations of our parameters corresponds to one *regression-testing strategy*. In the following, we denote a regression-testing strategy by [**RTC**, **NRT**, **NPR**, **RS**, **CR**] (e.g., [**MT**, **1**, **1**, **None**, **None**]) for short.

However, we do not regard the combination of **RS** $=$ **None** and **CR** $=$ **CR**, as using continuous reduction without an actual test-suite reduction is meaningless. Additionally, we do not regard the combination of **CR** $=$ **None** and **RS**! $=$ **None**, as the reduction of a test-suite for the same coverage criteria it was generated for (since the test suites of previous versions are ignored) is meaningless.

We, therefore, obtain 144 regression-testing strategies, which we will use to evaluate our regression-testing methodology. Additionally, two of these strategies are considered as *baselines*:

- Baseline 1 (basic regression testing strategy): [**MT**, **1**, **1**, **None**, **No-CR**].
- Baseline 2 (basic regression testing without previous test suites): [**MR**, **1**, **1**, **None**, **None**].

Subject Systems. We consider preprocessed C programs as our subject systems, which consist of an entry-function (i.e., the function under test). The program also needs to include all other functions having callee-dependencies to the function as well as all global variables and definitions needed to compile the program. Additionally, we require a version history of the program under test. We, therefore, focus on open-source projects from GitHub. Lastly, the subject systems need to fulfill the following requirements.

- The subject system must be processable by the test-case generator used in our framework.
- The function-under-test (as well as all callees) must have at least 4 different modifications throughout the version history.
- The signature of the function-under-test must contain input parameter which affect the return value (i.e., there exist different input values leading to different return values) or the value of at least one global variable. The existence of such values for the input parameters is checked using the test-case generation technique from Sect. 4.3 (which may timeout in which case we assume no such input values to exist).
- Calling the function-under-test multiple times with the same input values produces the same return value and global-variable values (i.e., no non-deterministic behavior or external system-function calls are considered).

The resulting collection of subject systems comprises program units from open-source projects published in GitHub:

- **betaflight**[2] contains six program units from the flight controller software *betaflight*.
- **netdata**[3] contains six program units from the infrastructure monitoring and troubleshooting software *netdata*.
- **wrk**[4] contains one program unit from the HTTP benchmarking tool *wrk*.

The subject systems have between 270 and 950 lines of code after removing unnecessary code (e.g., unused functions and variables, replacing aliases, etc.). The number of source code changes (i.e., commits that affect actual source code within the program unit) for each subject system ranges from 4 to 18.

Unfortunately, we could not use grbl (the controller software of the laser plotter in our example, see Sect. 2.1) as subject system. This is due to the fact, that the functions of grbl did not fulfill our requirements for our subject systems. The full list of our subject systems can be found in Appendix A.

[2] https://github.com/betaflight/betaflight
[3] https://github.com/netdata/netdata
[4] https://github.com/wg/wrk

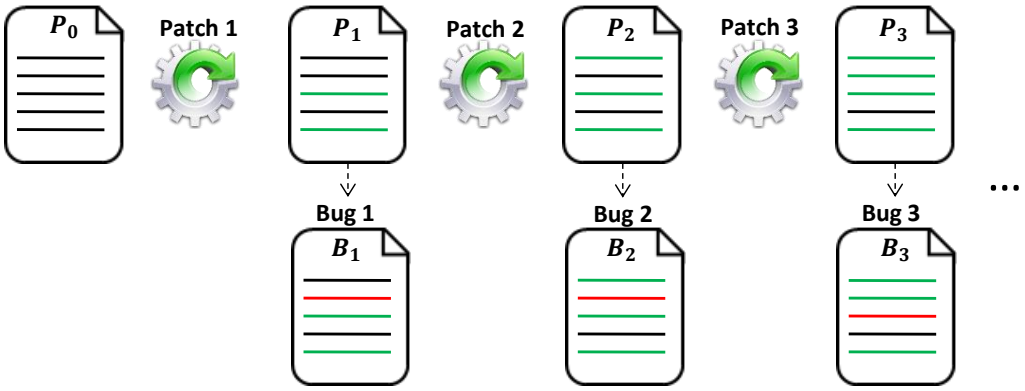


Figure 4.15: Bug Simulation using Mutation Testing [154]

Simulating Bugs. Although testing (e.g., regression testing) is crucial for software maintenance and quality assurance, software projects that provide a version history as well as properly documented information about real bugs are still scarce. To evaluate the interaction between program modifications, bugs, and the detection of the bugs by regression testing, both information are needed. Therefore, we rely on synthetically generated bugs, simulated by utilizing methods from mutation testing. As explained in Sect. 2.2.4, mutation testing is used to simulate common program faults made by developers [12]. This technique is well-established to measure the effectiveness of test-suites in terms of probability to detect real bugs. It has been shown that detecting mutants is related to the detection of real bugs [12].

Figure 4.15 provides an overview of our approach. First, we create mutants (using the tool MUSIC [143], a mutation tool for C programs) for all versions other than the initial version P_0 (i.e., a single mutant for each P_i with $i > 0$). Next, we generate the test-suite T_i based on the bugged (i.e., mutated) version B_i. Therefore, we simulated the bug to be included in the corresponding commit i. We check whether the bug has been detected by comparing the output (i.e., the return value and the values of global variables, see Sect. 4.3.1) of the bugged version B_i and the original version P_i. In case the results differ, the bug has been detected. To simulate the bugs, we utilize 62 different mutation operators (the full of mutation operators is provided in Appendix A). These operators can be grouped into three different types of operators. First, mutation operators replacing constants and variables by other constants or variables (e.g., replacing 5 with 10, or replacing variable a with variable b). Second, mutation operators that replace operators in the source code with other operators (e.g., replacing $>$ with $<=$). Third, we utilize mutation operators that replace pointer and array references (e.g., replace pointer $pt1$ with pointer $pt2$).

Data Collection. As described before, we first generate a bugged revision B_i for all versions $P - i$ other than P_i. Next, we create a new program or versions and previous versions (ie comparing B_i with $P_i - 1$, P_i with $P_i - 2$, depending on **NPR**). For each new program, we then create a comparator-program (see Sect. 4.3.1) for revealing test cases and a comparator-program for traversing test

cases. For traversing test cases, each changed line of code will be marked by a label that is used as test goal for the test-case generation. We also create a comparator-program for the original version P_i and the bugged version B_i to measure the effectiveness of the test-suite (i.e., if the test-suite manages to detect the bug). In some cases, no comparator-program can be generated due to technical limitations. In this case, the comparator-program will be skipped, and no test-suite will be generated.

Next, we apply the test-case generation (see Sect. 4.3) to generate test-suites for each bugged version using all 144 strategies.

To answer **RQ1**, we measure the effectiveness of the test suites. To this end, we check if at least one test case present in a test suite T_i^s generated for version i with strategy s manages to detect the bug B_i (i.e. leads to different output values for the bugged version compared to the original version, denoted as $P_i(tc) \neq B_i(tc)$). We denote the detection of a bug B_i by a test suite T_i^s as $detects(T_i^s, i) = 1$ if $\exists t \in T_i^s : P_i(t) \neq B_i(t)$, or 0 otherwise.

Therefore, we calculated the effectiveness of a test suite T_i^s as follows.

$$effectiveness(T_i^s) = detects(T_i^s, i) \tag{4.1}$$

The effectiveness of a strategy s is calculated as

$$effectiveness(s) = \frac{\sum_{i=1}^{n} effectiveness(T_i^s)}{n} \tag{4.2}$$

where n is the number of revisions.

To answer **RQ2.1**, we measure efficiency in terms of CPU time, denoted as $efficiency_{CPU}$, by aggregating the CPU time for all phases (i.e., the initialization phase of the CPAchecker, the reachability analysis, the test-case extraction, and the test-suite reduction). The initialization phase of CPAchecker is executed once per system under test. However, all other phases are executed multiple times based on the parameter **NRT**. Even though the test-suite reduction is also applied only once, the CPU usage is naturally subject to all parameters.

Test-case generation based on reachability analysis is undecidable and might take indefinitely long. Therefore, we use a global timeout after which the test-case generation terminates. For this reason, the number of test cases might be lower than specified by the parameter **NRT**, even in case that more potential test cases (i.e., not traversed paths for a test goal) might exist.

The efficiency in terms of CPU time ($efficiency_{CPU}$) for a strategy s is calculated as

$$efficiency_{cpu}(s) = \frac{\sum_{i=1}^{n} efficiency_{cpu}(T_i^s)}{n}. \tag{4.3}$$

To answer **RQ2.2**, we measure the number of test cases for each test suite. We then calculate the average for each test suite of a single strategy. Therefore, the efficiency in terms of the number of test cases ($efficiency_{size}$) is calculated as follows.

$$efficiency_{size}(T_i^s) = |T_i^s| \tag{4.4}$$

The overall efficiency of a strategy s is calculated as

$$efficiency_{size}(s) = \frac{\sum_{i=1}^{n} efficiency_{size}(T_i^s)}{n}. \tag{4.5}$$

Lastly, to answer **RQ3**, we calculate mean values for each strategy for both effectiveness and $efficiency_{CPU}$ and calculate the trade-off as follows.

$$trade\text{-}off(s) = \frac{effectiveness(s)}{efficiency_{size}(s)}. \tag{4.6}$$

Measurement Setup. All tools used throughout the evaluation were executed on an Ubuntu 18.04 machine with an Intel Core i7-7700k CPU and 64GB Ram. For practical reasons, we limited the CPU time of test-case generation to 900 seconds per program under test (as parameter **NPR** creates different programs, the timeout for the test-case generation of a strategy is 900 seconds times **NPR**). Additionally, we limited the coverage measurement (for test-suite reduction and effectiveness evaluation) to 30 seconds per test-case, to prevent indefinite execution of endless loops in the program under test. The test-suite reduction was not limited, as the CPU time needed was negligible. The test-case generation was executed with Java 1.8.0-171 and a Java heap size of 15GB. The test-suite coverage measurement was executed with Python 3.6.9.

4.6.3 *Results*

The results for **RQ1** are shown in Fig. 4.16a for all strategies with parameter **RTC** set to **MR**. Figure 4.16b shows the results for all strategies with parameter **RTC** set to **MT**. The results for **RQ2.1** with **RTC** = **MR** are depicted in Fig. 4.17a and for **RTC** = **MT** inf Fig. 4.17b. Figure 4.18a and 4.18b show the results for **RQ2.2** with the strategies with **RTC** = **MR** and **RTC** = **MT**, respectively. Lastly, the results for **RQ3** are shown in Fig. 4.19a for strategies with parameter **RTC** = **MR** and in Fig. 4.19b for strategies with parameter **RTC** = **MT**.

Remarks. The box plots (see Figs. 4.18a, 4.17a, 4.18b, and 4.17b) aggregate the results for our evaluation after applying the respective formulas 4.5 and 4.3 (see above) for all of our subject systems. The boxes in the box plots show the upper and lower quartile of the results, while the whiskers show the minimum and maximum values of the results (excluding outliers). Additionally, the dashes inside the boxes depict the mean values of the results. The plots for the effectiveness only show the mean values, as the effectiveness of a test-suite is either 1 or 0 making the use of box plots irrelevant. The plots for the trade-off values are single values, as they are based on the calculated mean values of effectiveness and $efficiency_{CPU}$.

RQ1 (Effectiveness). There are two strategies, namely - [**MR**, **3**, **3**, **None**, **No-CR**] and [**MR**, **2**, **3**, **None**, **No-CR**], which reach the best effectiveness with an average bug detection rate of 0.3659. The first baseline ([**MT**, **1**, **1**, **None**, **No-CR**]) reaches an average detection rate of 0.306, while the second baseline ([**MT**, **1**, **1**, **None**, **None**]) has a bug detection rate of 0.224. The worst strategy is [**MR**, **1**, **3**, **ILP**, **CR**] with an average detection rate of 0.146 bugs per program version.

RQ2 (Efficiency). The strategies with the best efficiency in terms of CPU time ($efficiency_{CPU}$) are both baseline strategies [**MT**, **1**, **1**, **None**, **No-CR**] and [**MT**, **1**, **1**, **None**, **None**] with an average CPU time of 6.058 seconds. The worst strategy is [**MR**, **3**, **3**, **ILP**, **No-CR**] with a mean value of 994,66 seconds per test-suite.

In terms of test-suite size ($efficiency_{size}$), the best strategies are [**MR**, **1**, **1**, **ILP**, **CR**], [**MR**, **2**, **1**, **ILP**, **CR**], [**MR**, **3**, **1**, **ILP**, **CR**], [**MR**, **1**, **1**, **RANDOM**, **CR**], [**MR**, **2**, **1**, **RANDOM**, **CR**] and [**MR**, **3**, **1**, **RANDOM**, **CR**] with 0.225 test cases per test-suite on average. The largest number of test cases per test-suite (i.e., worst $efficiency_{size}$) constitutes strategy [**MT**, **3**, **3**, **None**, **No-CR**] with 115.44 test cases on average.

RQ3 (Trade-Off). In terms of trade-off between effectiveness and $efficiency_{CPU}$, the best strategy is the first baseline [**MT**, **1**, **1**, **None**, **No-CR**] with an average ratio of 0.0504 bugs detected per second. The worst strategy is [**MT**, **1**, **3**, **ILP**, **CR**] with 0.0017 bugs per second.

4.6.4 *Discussion and Summary*

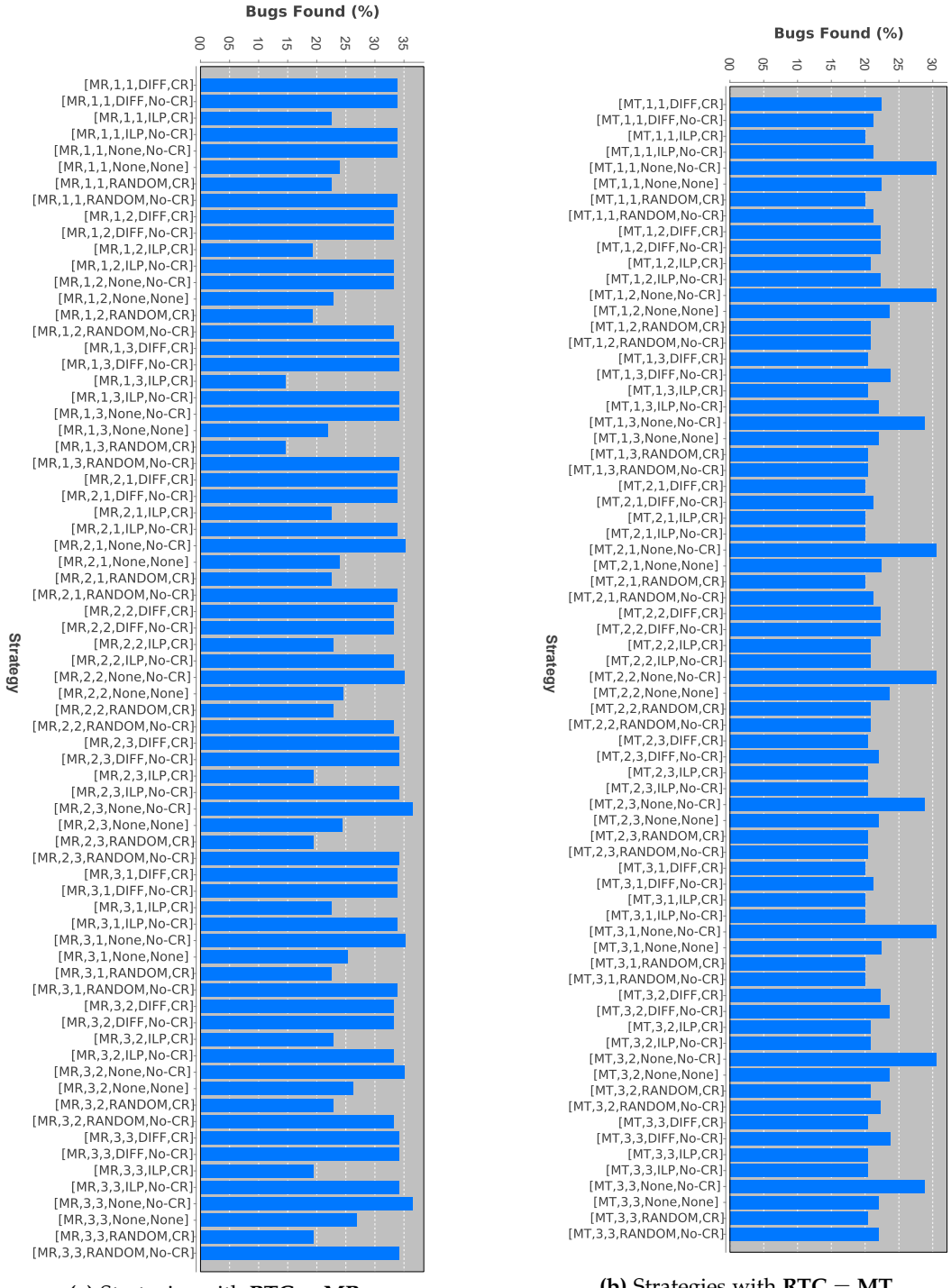
RQ1 (Effectiveness). When changing the parameter **RTC** from **MT** to **MR** the effectiveness increases for all strategies. In fact, nearly all strategies with parameter **RTC** set to **MR** are better than the best strategy with **RTC** = **MT**. Therefore, parameter **RTC** has the highest impact on effectiveness of all parameters. However, even though less than **RTC**, the other parameters have an impact on effectiveness, as well.

Strategies with parameter **RS** set to **None** have a higher effectiveness compared to strategies with **RS** $\neq$ **None**. This is due to the fact that test-suite reduction often does remove test cases that will detect the bug, as the test-suite reduction criteria do not correspond to the actual bug detection rate of a test case. Parameter **CR** only has a small impact for most strategies, however, combining **CR** = **CR** with **RS** = **ILP** causes a significant loss in effectiveness compared to other strategies. Lastly, both parameters **NRT** and **NPR** increase the effectiveness with increasing values, although with diminishing returns.

The best strategy is [**MR**, **3**, **3**, **None**, **No-CR**] (i.e., modification revealing test cases, three test cases per test goal, up to three previous revisions and no test-suite reduction) in terms of effectiveness. Compared to the first baseline [**MT**, **1**, **1**, **None**, **No-CR**] the increase in effectiveness is approximately 19%. Compared to the second baseline [**MT**, **1**, **1**, **None**, **None**] the increase is approximately 61%. Parameter **RTC** constitutes the highest impact on effectiveness.

RQ2.1 (CPU Time). The largest impact on effectiveness in terms of CPU time constitutes parameter **RTC**. In fact, choosing **MR** instead of **MT** increases CPU time by nearly 20 times. Parameter **NPR** increases CPU time nearly linearly based on the value selected (which was predictable, as increasing **NPR** increases the number of test-case generation executions).

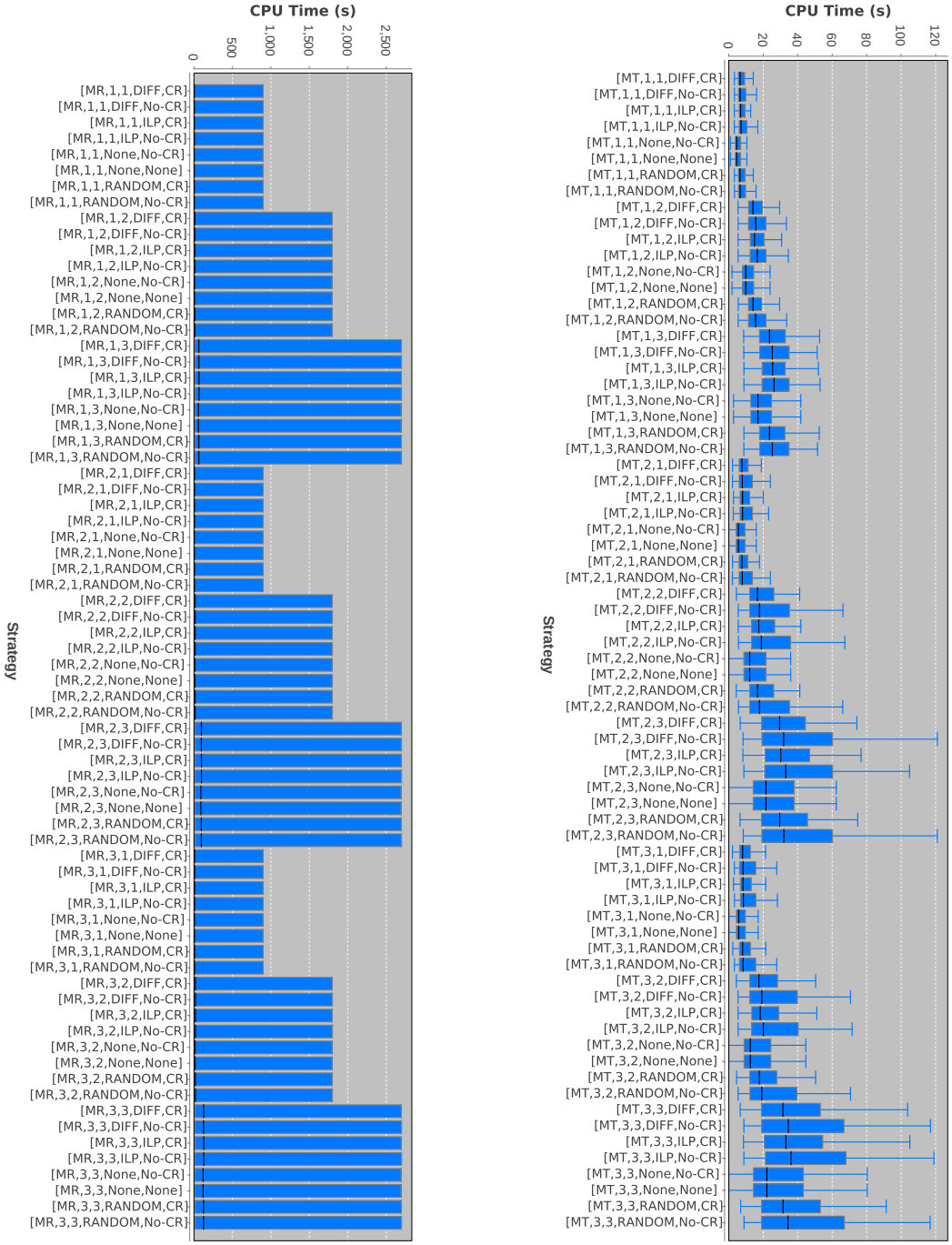
Surprisingly, parameter **NRT** has a small impact on CPU time. This is counter-intuitive, as usually, finding new paths for test-case generation is more expensive than the first path. However, this is most likely due to the fact that much



(a) Strategies with **RTC = MR**

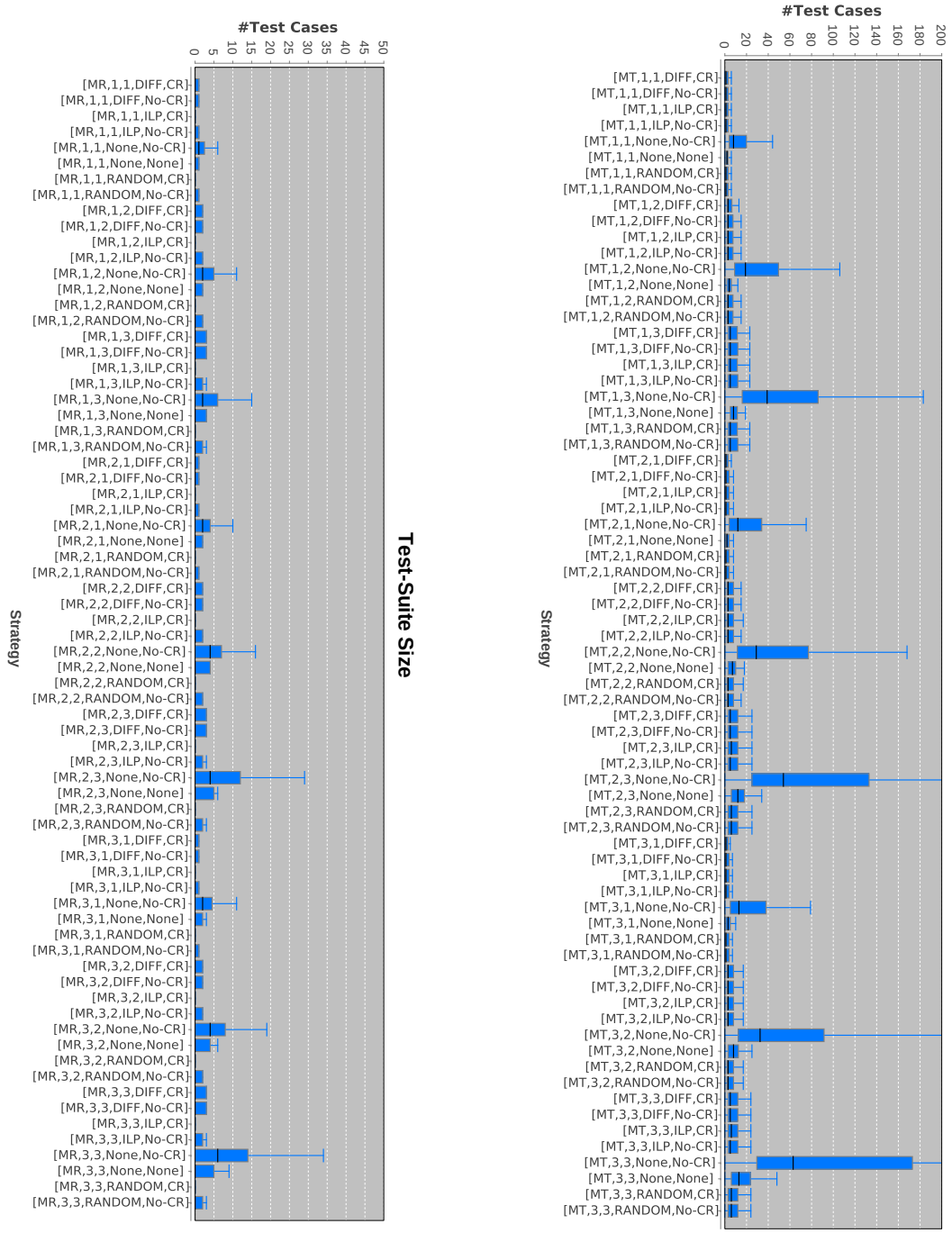
(b) Strategies with **RTC = MT**

Figure 4.16: Results Effectiveness [154]



(a) Strategies with **RTC = MR** **(b)** Strategies with **RTC = MT**

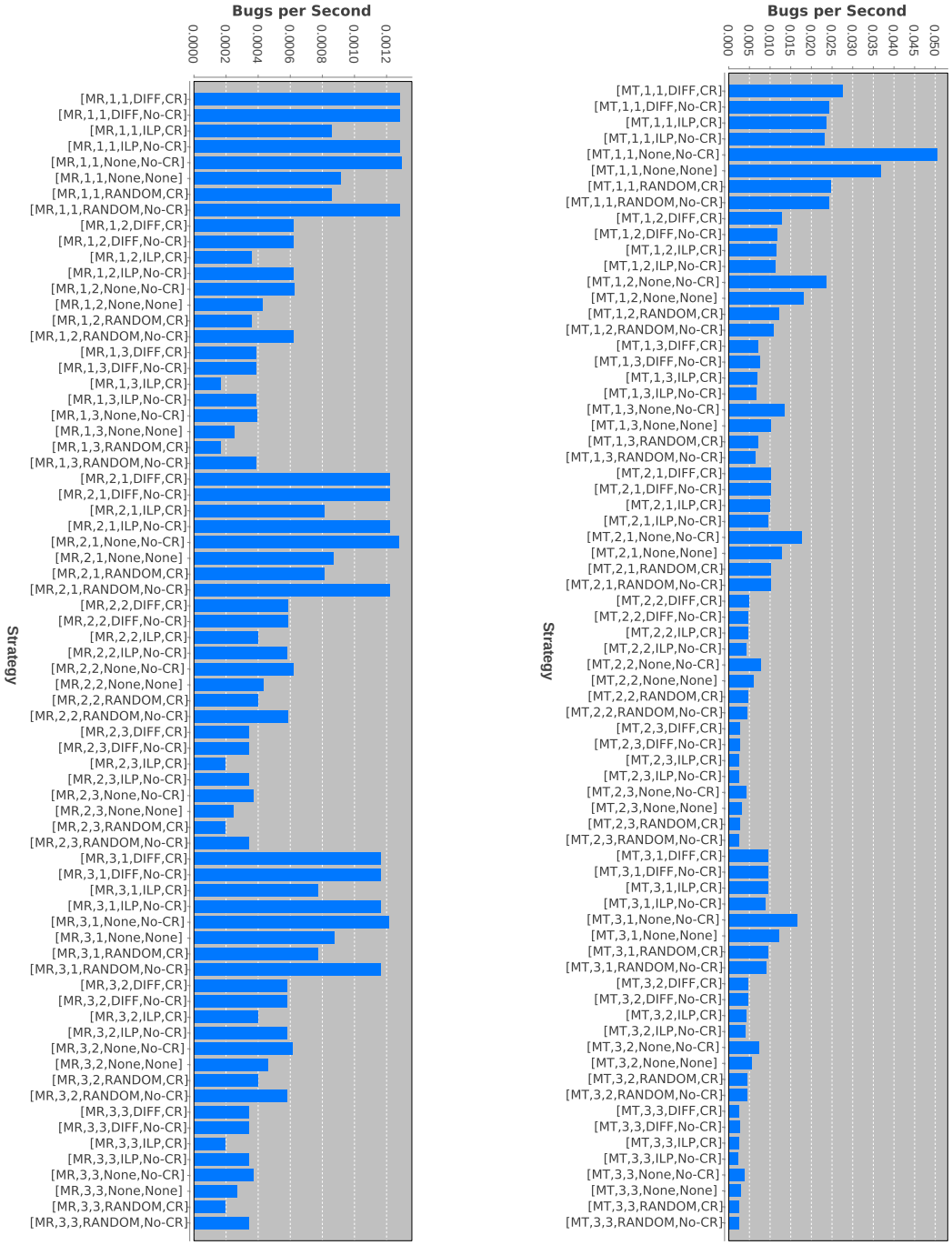
Figure 4.17: Results Generation Time [154]



(a) Strategies with **RTC = MR**

(b) Strategies with **RTC = MT**

Figure 4.18: Results Test-Suite Size [154]



(a) Strategies with **RTC** = **MR** **(b)** Strategies with **RTC** = **MT**

Figure 4.19: Results Bugs per Second [154]

information acquired during reachability analysis for the first test case also helps with the next test cases.

The impact of the remaining parameters **CR** and **RS** are negligible in terms of CPU time for all strategies with **RTC** = **MR** in terms of CPU time. However, if **RTC** = **MT** is selected, they do affect CPU time, although still only a small margin.

The strategies constituting the highest $efficiency_{CPU}$ are both baseline strategies [**MT**, **1**, **1**, **None**, **No-CR**] and [**MT**, **1**, **1**, **None**, **None**] (i.e., modification traversing test cases, one test case per test goal, one previous revision and no test-suite reduction, either ignoring or using the previous test-suite (as this makes no different in CPU time)) with an average CPU time of 6.058 seconds. Parameter **RTC** has the highest impact on CPU time by a large margin.

RQ2.1 (Test-Suite Size). The highest impact on $efficiency_{size}$ constitutes parameter **RS**. In case of **RS** = **None** the test suites grow for each new version and, therefore, grow continuously. In case of **RS** ≠ **None** the test suite are reduced at least once for the current version, therefore, reducing the amount of test case significantly. However, the impact of changing test-suite reduction techniques is small.

Parameter **RTC** has the second-highest impact on test-suite size. This is due to the fact that selecting **MT** leads to multiple test goals (i.e., one test goal per changed line of code), which usually leads to multiple test cases. However, selecting **RTC** = **MR** leads to one single test goal, which is guaranteed to show different output behavior (see Sect. 4.3.1).

Both parameters **NRT** and **NPR** have a nearly linear impact on the test-suite size corresponding to their respective values. Lastly, parameter **CR** has a small impact on test-suite size.

The best efficiency in terms of test-suite size have the strategies [**MR**, **1**, **1**, **ILP**, **CR**], [**MR**, **2**, **1**, **ILP**, **CR**], [**MR**, **3**, **1**, **ILP**, **CR**], [**MR**, **1**, **1**, **RANDOM**, **CR**], [**MR**, **2**, **1**, **RANDOM**, **CR**] and [**MR**, **3**, **1**, **RANDOM**, **CR**] (i.e., modification revealing test cases, one test case per test goal, one previous revision, **ILP** or **RANDOM** test-suite reduction strategy and using the reduced test-suite of the previous revision). Compared to the first baseline [**MT**, **1**, **1**, **None**, **No-CR**] the increase on $efficiency_{size}$ is approximately 6200%. Compared to the second baseline, $efficiency_{size}$ is increased by approximately 970%. The highest impact on the test-suite size has parameter **RS**.

RQ3 (Trade-Off). While there are parameters, which did not have any (or small) impact on either effectiveness or efficiency, all parameters have a large impact on the trade-off. This is due to the fact that each parameter has either impact on effectiveness or efficiency (or both). Therefore, the main driver of the impact on trade-off is actually the interaction of the parameters.

The best trade-off constitutes the first baseline [**MT**, **1**, **1**, **None**, **No-CR**] (i.e., modification traversing test cases, one test case per test goal, one previous revisions, no test-suite reduction and using the test-suite of the previous revision as well), which has a decent fault-detection but is extremely efficient in terms

of CPU time for test-case generation. This leads to a trade-off with an average of 0.0503 bugs detected per second. Compared to the second baseline [**MT**, **1**, **1**, **None**, **None**] the trade-off is improved by approximately 36%.

Remarks. Surprisingly, strategy [**MT**, **3**, **3**, **None**, **No-CR**] is less effective compared to strategy [**MT**, **3**, **2**, **None**, **No-CR**], which is counter-intuitive, as effectiveness is supposed to increase with increasing values of **NPR**. However, in some cases the comparator-programs for comparing bugged version B_i compared to P_{i-2} are invalid, while for P_{i-3} are valid. In this case, if the resulting test suite does not detect the bug, the mean value of the effectiveness might differ, as for strategies with **NPR** $= 2$ no test-suite exists, while for strategies with **NPR** $= 3$ a test-suite does exist with an effectiveness of 0. This is the reason for the discrepancy between the results of our evaluation and the theoretical specification of the design of our technique. However, only approximately 8% of the comparator-programs are non-valid. Therefore, it should not affect the results by a large margin. Additionally, note that for several strategies with **RS** $=$ **ILP** and **RS** $=$ **RANDOM** lead to the same $\mathit{efficiency}_{\mathit{size}}$. This is due to the fact, that for those strategies at most one test case is enough to fulfill the required coverage (or none, if the test goal cannot be reached by any of the test cases). Therefore, both will result in the same $\mathit{efficiency}_{\mathit{size}}$. Strategies with **RS** $=$ **DIFF**, however, will always select exactly one test case and, therefore, might lead to larger test suites.

4.6.5 *Threats to Validity*

Internal Validity. A threat to our internal validity is our assumption that all external factors (e.g., platform, environment, etc.) remain constant during the evaluation. Only the source code of the system under test and the input values of the test cases might change. This assumption is called *controlled-regression-testing assumption* and is commonly used during regression testing [181]. Therefore, the assumption should not harm our validity. We also do not provide proof of the soundness of our methodology. However, we tested the test-case generation loop manually by checking the results for selected subject systems. Still, test-case generation, in general, is undecidable and, therefore, we cannot be certain that we found all possible test cases needed to fulfill our criteria (e.g., due to timeouts or imprecise counterexamples). Nevertheless, even if we increase the precision of our technique and find more test cases that satisfy our criteria, the results should stay the same (at least relatively). Additionally, our selection of mutation operators might be a threat to internal validity. However, we carefully selected mutation operators leading to useful (for our regression scenario) resulting mutants (i.e., only affecting one line of code, no code deletion, resulting in compilable programs, etc.). Therefore, our selection of mutation operators should not harm the validity of our results. We also limit our evaluation to functional testing only at unit level. This might threaten internal validity. However, unit testing is one of the most established testing techniques in practice. Additionally, unit testing provides a decent starting point to evaluate our technique, as the subject systems, the interaction of parameters, and the CPU time needed are still graspable compared to

other testing levels. Nonetheless, our concepts can easily be used for integration or system testing.

One downside of the current implementation of our technique is the missing functionality to check whether the previous test-suite already fulfills the criteria of the strategy. This might affect CPU time, as test-case generation is, in general, more expensive compared to coverage measurement. We plan to add this re-usability of test suites into our framework. This might increase $efficiency_{CPU}$ for some strategies and, therefore, also affect the trade-off. However, we do not expect fundamental changes, as the effectiveness and $efficiency_{size}$ will stay the same.

Lastly, we currently only support C programs. Still, this should not harm the validity of our results, as c is still one of the most prominent languages used. Additionally, we expect the results to be very similar for other imperative programming languages (e.g., most object-oriented languages).

External Validity. The main threat to the external validity of our evaluation is the missing comparison to other tools. However, as far as we know, no competitive tool for RegreTS exists. Particularly, concerning the configurable number of test cases, especially in combination with modification revealing test cases.

Surprisingly, we also did not manage to use other test-case generators for strategies with **RTC** = **MR**, which should (per design) have been possible. The most probable reason is the subject systems we selected. All subject systems are real-world projects, which we checked to work with our test-case generator. However, other tools might not be able to handle the subject systems due to multiple reasons (e.g., not supported language constructs). We tried to use different test-case generators (i.e., Klee, FuSEBMC, Symbiotic and PRTest) from the International Competition on Software Testing (Test-Comp[5]). Unfortunately, most test-case generators were either barely able to generate test cases for our subject systems or only provided irrelevant test cases (e.g., dummy test cases, or test cases that did not reach the specified test goal), therefore leading to empty test suites after the test-suite reduction step. Only PRTest was able to generate useful test cases. However, due to timeouts only for less than half of our subject systems. However, the focus of our work is the comparison of strategies, not the comparison of different test-case generation techniques. As the strategies should provide similar results when executed on different test-case generators, we expect no fundamental changes of our insights, as the relative behavior of the strategies should stay similar.

Another threat to our external validity is the selection of subject systems and the generation of synthetic bugs. Unfortunately, real-world systems with sufficient information about version changes and existing (or fixed) bugs are scarce. We checked multiple prominent repositories for regression testing. First, we checked CoreBench. However, the subject systems were usually not at unit level, and the version histories were rather short. Second, we checked the subject systems from the *regression-verification tasks*, which is part of the SV-Benchmarks [22]. However, the version histories were also very short and, therefore, not relevant for our methodology. Additionally, some of the subject systems were not self-contained

[5] https://test-comp.sosy-lab.org/2021/

and, therefore, unusable for our technique. Lastly, we also checked the SIR repository [61], which provides information of bugs from prominent systems (e.g., make, gzip, etc.). However, these systems either could not be handled by our test-case generator for technical reasons (e.g., unsupported language constructs, for example micro-controller specific code) or unusable (e.g., not self-contained). Nevertheless, mutation testing has been shown to be a reliable technique for fault-injection for effectiveness measurement of test-suites [12]. Additionally, our subject systems are real-world programs from different projects and different domains. Therefore, we do not expect our results to change fundamentally if evaluated on non-synthetic bugs.

Lastly, we rely on third-party software, namely CPAchecker, Music, and Test-Cov. All three tools are established and have been used for other experiments in the recent past. CPAchecker and Test-Cov are even used in the International Competition on Software Testing (Test-Comp) for the last 3 years. Therefore, we expect them to produce sound results.

4.7 Related Work

In this section, we discuss related work in regression testing, testing techniques for change-impact analysis (i.e., differential/revealing test-case generation and mutation testing), and, lastly, regression verification.

4.7.1 *Regression Testing*

A comprehensive overview of different regression-testing strategies is provided by Yoo and Harman [181]. In this work, they describe three different categories of regression testing:

1. *minimization* of test suites
2. *selection* of test cases
3. *prioritization* of test cases

Test-Suite Reduction is concerned with the removal of redundant test cases from test-suites based on a given criterion. The goal is to reduce the execution time of test suites during regression testing. As test-suite reduction can be reduced to the set-cover problem, it is NP-hard. Therefore, many works propose different heuristics to reach a near-optimal solution for this problem [46, 88, 99, 136]. Those techniques require as input an *existing* test suite and a metric to measure effectiveness of test cases. All approaches for test-suite reduction used in this work have already been proposed. The greedy algorithm used by *DIFF* was introduced by Beyer and Lemberger in [29]. The idea to use ILP solvers for test-suite reduction to compute an optimal solution was initially proposed by Khalilian and Parsa in [109]. The FAST++ algorithm has been introduced by Crucian et al. [53]. However, these work focus on single revisions (i.e., two subsequent versions) and do not investigate the interaction between test-case generation tailored for regression testing and the test-suite reduction techniques.

There also exists other work investigating the impact of test-suite reduction effectiveness of real systems. Shi et al. use build failures as effectiveness measurement and evaluate existing test-suite reduction techniques, whether the reduced test suite still manages to detect the build failure beforehand [164].

Chan et al. proposed a new technique that uses assertion coverage (i.e., test cases that will fail an assertion), instead of usual coverage-based criteria, to further improve effectiveness of the reduced test suites [45].

In previous work, Shie et al. evaluated the combination of test-suite reduction and test-case selection to increase efficiency in terms of test-suite size of regression testing [163]. However, none of these works consider the additional parameters evaluated in this work. For example, we also consider the number of test cases per test goal and the number of previous revisions taken into account. Additionally, we provide means to choose between modification traversing and modification revealing test cases to provide an insight between efficiency and effectiveness of the regression test criterion. All those strategies have been shown to impact effectiveness and/or efficiency. Especially the trade-off has been subject to the interplay of all parameters. Additionally, those works do not consider generating multiple test cases per test goal which we have shown to increase effectiveness even further.

Test-case selection aims at selecting test cases from an existing test suite specifically for a specified version of the program under test. The idea is to create a subset of test cases that will be executed on a new program version without loss of effectiveness. Different techniques have been subject to research. For example, control-flow analysis [91, 89, 90] and data-flow analysis [81, 87, 86, 168]).

In 2018, Wang et al. used a catalog of program refactorings and identified test cases affected only by these behavior-preserving modifications. These test cases were not selected for retesting, as they would test only the behavior-preserving modifications (and therefore unable to detect any newly introduced bugs) [177].

Choudhary et al. use multi-objective optimization techniques for test-case selection. They try to minimize the number of uncovered test goals and maximize the number of uniquely covered test goals (i.e., covered by only one test case) under a fixed test-suite size. Both optimization conditions are used to compute a Pareto-front of optimal solutions [49].

Marijan and Liaaen propose a technique for test-case selection for highly-configurable software. They combine metrics for measuring the degree of overlapping test cases (i.e., being related to similar configuration options) with historical data on the effectiveness of test cases. Based on those metrics, they identify inefficient redundancy of test cases, which are then not selected for regression testing. [129]

However, none of these works aim at generating modification revealing test cases to further increase effectiveness during regression testing. Additionally, most works only take a single revision into account, especially not multiple previous versions, as supported by parameter **NPR**. Lastly, they do not study the interaction between different regression testing configurations as done in our methodology. Additionally, those works do not consider generating multiple test cases per test goal. Therefore, both of our parameters **NPR** and **NRT** are not considered by

these works and we have shown that they can improve effectiveness in terms of bug detection for regression testing.

Test-case prioritization focuses on the execution order of existing test cases to detect as many bugs as fast as possible [179]. To idea is to compute fault detection probabilities for test cases, as well as their execution time (e.g., from previous executions) to create an execution order. Usually, a code coverage criterion is used as effectiveness measure to statically compute an execution order for the test cases [151, 152, 66, 67, 128, 153]. In 2016, Wang and Zeng proposed a dynamic prioritization technique based on actual fault-detection from previous test-case executions and other properties based on historical data of the system under test [178].

For our methodology, test-case prioritization is currently out of Scope. However, it might be integrated for test-case generation to focus on the execution of test-cases more likely to detect bugs.

Most existing regression testing techniques are not concerned with test-case generation to further improve effectiveness during regression testing. The focus of those works is to optimize the efficiency with the same or similar effectiveness based on an existing set of test cases. Additionally, most works on regression testing focus on a single revision instead of full version histories. Therefore, our parameters **NPR** and **NRT** are out of scope for these works and we have shown that they do improve effectiveness in terms of bug detection during regression testing.

4.7.1.1 *Regression-Test-Case Generation.*

Different works are concerned with test-case generation tailored for regression testing, similar to our methodology. In the following, we will discuss those works.

In 1998 McKeeman proposed the idea of differential testing, with the following problem statement [132]. Given two comparable programs and a set of existing test cases, the programs can be checked for bugs by running the test cases. If the outputs differ, the test execution loops indefinitely, or the execution crashes, the test case is likable to be a bug-revealing (to actually detect a bug) test case. Based on this problem statement, Evans and Savioa proposed an approach to use the original system and a modified system and two test-suites for both systems. Those test suites are then used to check for different behavior of the original and the modified system. All test cases leading to different output behavior are then used for regression testing [70].

Korel and Al-Yami initially proposed automated generation of test cases for revealing differences between two (Pascal-based) program versions for regression testing [111]. CSmith is a tool developed at the University of Utah that combined differential testing with grammar-based fuzzing to generate compilable C programs to find bugs in c-compiler implementations [180].

Taneja and Xie proposed an approach for differential test-case generation to compare two versions of a Java program. To this end, they create a comparator-program similar to the comparator-program introduced in Sect. 4.3.1. In case of a different output behavior, they use failing assertions, enabling them to use

off-the-shelf test-case generators that support assertion coverage for Java programs [169].

Still, those works differ from our methodology, as the focus is on two subsequent program versions instead of a configurable number of previous versions. Additionally, they do not support test-suite reduction or multiple test cases per test goal. Therefore, they do not study the interaction between those parameters, which we have shown to impact effectiveness and efficiency.

4.7.1.2 *Mutation Testing.*

Mutation testing is (originally) used to measure effectiveness of existing test-suites. To this end, multiple syntactically changed *mutants* (i.e., to generate synthetic bugs) are derived from an original program, based on different mutation operators (i.e., by simulating programmer-errors) [103]. A test case detects a mutant if the observable behavior of the mutant when executed with the test case differs from the original program. Multiple techniques have been proposed to generate test cases for mutation detection. For example, Fraser et al. proposed an approach to generate test cases tailored for mutation detection by utilizing genetic algorithms to generate test cases guaranteed to detect a given set of mutants [74]. In 2011, Harman et al. proposed a technique that combines symbolic execution and search-based heuristics to identify test cases likely to detect mutants (although not guaranteed) [84]. Souza et al. use hill climbing to generate test cases to detect mutants [165].

Most of those approaches use different heuristics to generate test cases. Therefore, they do not guarantee to reveal behavior changing program modifications. Additionally, those approaches do not support test-suite reduction, multiple test-cases per test goal (e.g., detection of a mutant), or version-history aware regression test-case generation (as supported by our parameters **NPR** and **NRT**). However, those technique might be incorporated into our framework to increase $\mathit{efficiency}_{CPU}$ of test-case generation (potentially reducing effectiveness).

4.7.2 *Regression Verification*

Regression verification aims to check whether a specification already verified for a program version is still valid for a subsequent program version. To this end, (partial) verification results are re-used to increase $\mathit{efficiency}_{CPU}$ during verification [97, 167]. Beyer et al. proposed an approach to re-use intermediate results (called abstraction precisions) for verifying later program versions [34]. Bohme et al. used input-space partitions to gradually verify subsequent program versions instead of fully verifying the program version for all possible input values [37]. In 2012 Chaki et al. lifted regression verification techniques to support multi-threaded programs [44]. Different works also exist to further increase regression verification $\mathit{efficiency}_{CPU}$ by applying state-space partitioning [17] or by improving the encoding of reusable verification information [71].

However, none of these approaches investigate the impact on effectiveness and $\mathit{efficiency}_{CPU}$ and $\mathit{efficiency}_{size}$ of regression-testing strategies or the interplay of

different regression testing configurations (e.g., test-suite reduction, multiple test cases, etc.) as in our methodology.

Lastly, there exists related work for test-case generation that was already discussed in Sect. 3.5.

4.8 Conclusion and Future Work

Conclusion. In this chapter, we presented a configurable methodology for regression testing with the focus of revealing bugs in evolving programs. Our methodology supports test-suite reduction, multiple test-case generation techniques, different regression-test-case criteria, and a configurable amount of previous versions taken into account. We currently support unit testing of C programs. The results of our experimental evaluation show that all parameters either affect effectiveness and/or efficiency. Additionally, all parameters affect the trade-off between effectiveness and $\mathit{efficiency}_{CPU}$ and, therefore, the strongest impact on the trade-off is the interplay of those parameters.

In conclusion, our experimental results show that to achieve the best $\mathit{efficiency}_{CPU}$ and the best trade-off modification traversing test cases should be used without test-suite reduction. However, the best effectiveness is reached by using modification revealing test cases, with multiple test cases per test goal and multiple prior revisions taken into account. Lastly, we have shown that in our experimental evaluation the test-suite reduction techniques *ILP*, *DIFF* and *RANDOM* obstruct effectiveness. Therefore, for optimal effectiveness in our setting those test-suite reduction techniques should not be used.

Future Work. In future works, we plan to extend our approach in multiple ways. First, we plan to improve the implementation of our test-case generation to support additional subject systems to extend our evaluation. For example, by supporting arrays as input parameters or return values. Additionally, we would like to incorporate different test-case generation techniques to improve $\mathit{efficiency}_{CPU}$ of the test-case generation (e.g., by combining fuzzing with hybrid-model checking). Next, we plan to use different test-case generation tools to generate test cases based on our strategies (as far as supported by the other tools) and compare results. We would also like to incorporate re-use of existing test-cases to increase $\mathit{efficiency}_{CPU}$ and $\mathit{efficiency}_{size}$ of our technique. Additionally, we would like to take different kinds of bugs into account (e.g., control-flow bugs and data-flow bugs) and measure the impact of our parameters on those bugs. Furthermore, we plan to use different coverage criteria for test-case generation. For example test-cases reaching the modification and propagating the change, however, not necessarily revealing it. This coverage criteria would provide a trade-off between effectiveness and $\mathit{efficiency}_{CPU}$ between modification traversing and revealing test cases. Finally, we plan to use RegreTS for different testing scenarios (e.g., integration testing or system testing) and to support different programming languages for test-case generation to extend the set of possible subject systems.

5

TESTING OF SOFTWARE-PRODUCT LINES

In the previous chapters, we were concerned with testing products, either as single products or as products within a version history. However, implementing software as a single product is often insufficient. Primarily in Industrie 4.0, many similar machines exist which share common core features and usually differ only slightly. Creating a single project (with implementation artifacts etc.) for each machine individually would strongly increase maintenance costs (see Sect. 2.5). A commonly used solution for this problem is to implement a so-called software-product line (SPL).

An SPL consists of a common code base within the solution space being customizable (usually by selecting and deselecting features, see Sect. 2.5). Usually, an SPL is accompanied by a configuration model that specifies the software variability within the problem space (often represented as a feature model). Lastly, the SPL has a mapping between the configuration model and the codebase. The mapping might be implicit by using the same feature names in the model and the codebase. [51]

However, real-world SPLs often allow for thousands of different configurations. Therefore, testing of all products becomes impossible due to the combinatorial-explosion problem. [62]

In this chapter, we will study the effectiveness and efficiency of different strategies for sample-based SPL testing. To this end, we will introduce methods developed in this thesis to measure effectiveness based on injected faults (i.e., mutants). The following research challenges will be tackled in this chapter.

- How to increase effectiveness of SPL testing in terms of potential bug detection ratio (**C1**).
- How to increase efficiency of SPL testing in terms of the number of products under test (**C2**).

In this chapter, we will first extend our running example to further motivate and illustrate different concepts. Next, we will first give an overview of existing state-of-the-art techniques in testing and test-case generation of SPLs.

In Sect. 5.4, we will present a novel approach, developed in the course of this thesis, to measure the effectiveness of strategies for sample-based SPL testing.

Next, we will show the implementation and the evaluation of our methodology. Lastly, we will discuss related work and give a summary and show possible future work.

The content of this chapter is based on the following publication:

> [155] Sebastian Ruland, Lars Luthmann, Johannes Bürdek, Sasche Lity, Thomas Thüm, Malte Lochau and Márcio Ribeiro. Measuring Effectiveness of Sample-Based Product-line Testing In *Generative Programming: Concepts & Experiences*, ACM, 2018.
> doi: 10.1145/3278122.3278130.

5.1 Running Example

As shown in Sect. 2.5, we encoded our running example as SPL for the remainder of this chapter. To ease the reading flow, the SPL version of our running example and the corresponding feature model is shown in Fig. 5.1 and Fig. 5.2 again. For testing SPLs, we annotate the test cases with their feature conditions, for which they are valid. For example, test case $t = \{([1], 0), -1\}(ISEMPTY)$ is valid for all products that have the feature $ISEMPTY$ selected. If the feature $ISEMPTY$ is not selected, the return value is -2 instead of -1, and, therefore, the test case would not be valid. For products that do not select the feature $ISEMPTY$ the test case $t = \{([1], 0), -2\}(!ISEMPTY \wedge !COUNT)$ can be used.

The same applies for the detection of bugs. In the current version of our running example there still exist three bugs. Namely Bug 1 in line 22 (i.e., incorrect assignment of 0 instead of 1 to variable *startIndex*), Bug 2 in line 18 (i.e., $x[0] - 2$ instead of $x[0] - 1$) and Bug 3 in line 27 (i.e., incorrect condition $x[i] <= y$ instead of $x[i] == y$). Bug 3 is a bug within the core of the SPL implementation and, therefore, detectable on all products. However, depending on the product, a different test case needs to be used. For example $t = \{([3, 2, 1], 1), 2\}(FIRST \vee LAST)$ detects Bug 2 on all products that have either feature $FIRST$ or $LAST$ selected (selecting both in forbidden due to the feature model, see Fig. 2.15). Bug 1 is only detectable in products that have feature $LAST$ selected (since the source code containing the feature is only present if $LAST$ is selected). For example, test case $t = \{([1], 1), -2\}(LAST)$ would detect Bug 2.

5.2 Sample-based Product-Line Analysis

As shown in the running example, bugs of SPLs are sometimes only detectable in certain products. However, exhaustive testing of an SPL P_{spl} would require to execute the tests on *every* program configuration $p_c = [\![P_{spl}, c]\!]$ corresponding to a valid product configuration $c \in C$ a (configuration-specific) test suite $T_c \subseteq T_{p_c}$. Testing an SPL in a product-by-product manner is, in practice, not feasible. This is due to the fact that the number of possible product configurations C grows exponentially in the number $|F|$ of features, thus leading to the well-known *combinatorial-explosion problem* (i.e., every additional optional feature in F may double the number of configurations in C). [172, 139, 68, 55, 115, 63]

Additionally, the high amount of similarity among different program configurations potentially leads to a high number of *redundant test-case executions* (i.e., given two configurations c, c', the number of common test cases in the intersection $T_{p_c} \cap T_{p'_c}$ is presumably very high).

```
1  int c;
2  int startIndex;
3  int lastIndex;
4  int increment;
5  int last;
6
7  int find_configurable (int x[], int y) {
8    #if defined(COUNT)
9    c = 0;
10   #else
11   last = -2;
12   #endif
13   #if defined(ISEMPTY)
14   if(x[0] <= 1)
15     return -1;
16   #endif
17   #if defined(FIRST) || defined(COUNT)
18   startIndex = x[0]-2;
19   lastIndex = -1;
20   increment = -1;
21   #elif defined(LAST)
22   startIndex = 0;
23   lastIndex = x[0]-1;
24   increment = 1;
25   #endif
26   for (int i = startIndex; i != lastIndex ; i += increment){
27     if (x[i] <= y)
28       #if defined(COUNT)
29       c++;
30       #else
31       last = i;
32       #endif
33   }
34   #if defined(COUNT)
35   return c;
36   #else
37   return last;
38   #endif
39 }
```

Figure 5.1: SPL Version $P_{0_{SPL}}$ of a Program Unit (adapted from [155, 154])

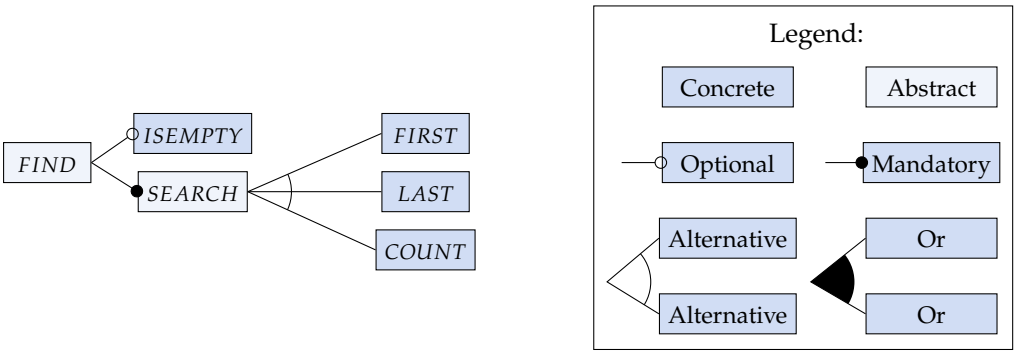


Figure 5.2: Feature Model of the Running Example in Fig. 5.1 (based on [155])

To tackle this issue, recent research is concerned with developing *efficient* strategies to reduce the computational effort of product-by-product analysis while trying to retain high *effectiveness* [172, 139, 68, 55, 115, 63].

Sample-based analysis constitutes one of the most established approaches in practice. [172, 63, 126] A *sample* $S \subseteq C$ is a preferably small, yet sufficiently diverse subset of *test configurations* to be analyzed in a product-by-product manner instead of the entire set C. A widely used sampling criterion is τ-wise combinatorial feature-interaction coverage requiring every valid combination of selections and deselections of τ features to be contained in at least one test configuration $c \in S$. [63, 126]

> **Example 5.1.** The sample $S = \{c_1, c_4, c_5\}$ (see Tab. 2.2) satisfies 1-wise combinatorial feature-interaction coverage on our running example as all valid selection- and deselection-decisions for every single feature are covered by at least one configuration in S. Hence, sample S is able to find all bugs of our running example as a RIPR test case for the respective bug can be executed on test configuration p_{c_2} with $c_2 \in S$. However, there also might exist bugs only detectable by products with specific feature interactions. For example, if we introduce a new bug by inserting `c++;` before line 16 the bug would only be detectable by selecting both *ISEMPTY* and *COUNT* in the same product. Therefore, it would not be detectable by our sample S. Instead, a sample S' satisfying 2-wise (pairwise) coverage is guaranteed to contain at least one test configuration on which this mutant is detectable since all combinations of both features must be present in the sample (in our running example, the sample would cover all possible configurations).

One of the most established sampling heuristics is to apply pairwise combinatorial feature-interaction testing (i.e., $\tau = 2$) [63, 126]. This technique relies on the assumption that most bugs in SPLs are caused by erroneous interactions among at most two configuration options (so-called *features*) [112]. However, increasing τ does not always increase effectiveness, since the feature interactions within the implementation also plays a vital role (e.g., if there exists no feature interaction $\tau = 1$ will always result in 100% effectiveness). Thus making any a-priori assessment of the expected effectiveness of different sampling criteria a challenging task.

5.3 Family-based Test-Case Generation

In this section, we will describe existing techniques for test-case generation for SPLs. Additionally, we will introduce a novel technique utilizing the negated paths from Sect. 4.3.

As described in the previous section, test-case generation product-by-product is usually infeasible due to the exponential growth of products compared to the number of feature variables.

Testing the SPL in a family-based manner allows a single test-case generation process to generate a full test-suite containing for each valid product a test case for each test goal that is reachable. The test goals are also annotated with presence

conditions (i.e., formula over feature variables for which the test case is valid), which is necessary for the novel approach in Sect. 5.4.

However, most test-case generation tools do not support compile-time variability (i.e., **#if** blocks, etc.) [108]. To this end, prior work has proposed a technique to lift compile-time variability to run-time-variability by replacing **#if** and other preprocessor directives by normal code (i.e., **if** blocks)[108]. This replacement is not always straightforward and might need to copy some parts of the source code to be compilable afterward. However, lifting compile-time variability to run-time-variability is always possible. Lifting the implementation of our running example with compile-time variability to run-time-variability would result in the running example in Fig. 5.3. Note that for run-time variability the features (i.e., *COUNT*, *ISEMPTY*, etc.) need to be included as variables. However, for simplicity reasons we omit them in the running example.

In case of an SPL with run-time variability, we can use the implementation for test case generation again. However, as we don't want to generate a test case on any configuration for each test goal but a test case for each configuration on that the test goal is reachable, we need to enhance the test case generation approach. To this end, Bürdek et al. [40] proposed a technique to use software model checking to generate test cases for each test goal on each configuration using a so-called blocking-clause. The idea is to execute a reachability analysis for a single test goal. Upon reaching the test goal, the presence condition (i.e., the formula over feature variables to reach the test goal) is computed. Afterward, the reachability analysis is started again. However, the presence condition is negated and added to the initial condition.

For our running example, the source code would first be transformed into a CFA. Figure 5.4 shows the resulting CFA. Note that the feature edges are encoded as dashed edges while code-edges are encoded as filled edges. During reachability analysis, we now include another abstract state to track the presence condition. Additionally, the path condition does not track feature variables since they are already tracked by the presence condition.

Example 5.2. Consider the edge $(7) \xrightarrow{!(x[0] <= 1)} (9)$ as test goal. Next, we will build an abstract reachability graph until reaching the test goal (see Fig. 5.5, the first substate denotes the location, the second substate denotes the path condition, and the third denotes the presence condition). Next, we extract the presence condition from the target state 9 (i.e., $COUNT \wedge ISEMPTY$) and add it to the test case covering the test goal (e.g., $t = \{([1], 0), -\}(COUNT \wedge ISEMPTY)$). Now we restart the reachability analysis. However, we initialize the initial presence condition with the negated presence condition of the test case (see Fig. 5.6 state 1). Next, the reachability graph is computed as before. When reaching state 7, the presence condition is now $false$ (due to $COUNT \wedge ISEMPTY \wedge !(COUNT \wedge ISEMPTY)$) in contrast to the first reachability analysis. Therefore, we continue with state 7′ and reach state 9 with another presence condition, from which we can now compute the next test case (e.g., $t = \{([1], 0), -\}(!COUNT \wedge ISEMPTY)$). Now, we could start the reach-

```
1  int c;
2  int startIndex;
3  int lastIndex;
4  int increment;
5  int last;
6
7
8  int find_configurable (int x[], int y) {
9    if(COUNT){
10     c = 0;
11   }else{
12     last = -2;
13   }
14   if(ISEMPTY){
15     if(x[0] <= 1)
16       return -1;
17   }
18   if(FIRST || COUNT){
19     startIndex = x[0]-2;
20     lastIndex = -1;
21     increment = -1;
22   } else if(LAST){
23     startIndex = 0;
24     lastIndex = x[0]-1;
25     increment = 1;
26   }
27   for (int i = startIndex; i != lastIndex ; i += increment){
28     if (x[i] <= y)
29       if(COUNT){
30         c++;
31       }else{
32         last = i;
33       }
34   }
35   if(COUNT){
36     return c;
37   }else{
38     return last;
39   }
40 }
```

Figure 5.3: Initial SPL Version $P_{0_{SPL}}$ with Run-Time Variability

ability analysis for a third time with the initial presence condition $!(COUNT \wedge ISEMPTY) \wedge !(!COUNT \wedge ISEMPTY)$. However, since the presence condition is equal to $false$ we would not create any states. Therefore, we have created test cases for test goal $(7) \xrightarrow{!(x[0] <= 1)} (9)$ for all products on which it is reachable.

Additionally, Bürdek [39] proposes multiple enhancements to this technique (e.g., path-sensitivity for state-merging, multi-property checking for SPL test-case generation, etc.). Those solutions are tailored for test-case generation with test-goal automata, which can encode even complex test-goals (e.g., FQL-Statements [98]). However, test-case generation with test-goal automata is expensive in terms of CPU time and memory usage. Due to this reason, we focused on test-goals encoded as lists of CFA edges in this thesis.

Therefore, we developed a novel technique for test-case generation of SPLs based on test-goals consisting of lists of edges, and, therefore, also increasing the $efficiency_{CPU}$ of the test-case generation process.

To this end, we used the path negation introduced in Sect. 4.4 and enhanced the path negation to support SPLs. The idea is quite similar to the path negation. However, for SPLs, we only focus on feature edges and ignore other branching edges.

Example 5.3. Consider again the edge $(7) \xrightarrow{!(x[0] <= 1)} (9)$ as test goal. Next, we will build an abstract reachability graph until reaching the test goal (see Fig. 5.7, the first substate denotes the location, the second substate denotes the path condition, and the third denotes the state of the negated paths.). In the first iteration, no negated path exists. Therefore, the state of the negated path remains empty. Note that we now include feature variables into the path condition, just like normal variables. When reaching a test goal, we compute a counterexample (Fig. 5.7 the path indicated by red edges). From this counterexample, we build the presence condition for the test case by conjugating all feature expressions on the path. For the given path, the presence condition is $COUNT \wedge ISEMPTY$. Additionally, we use the counterexample to compute valid parameter assignments to construct a test case (e.g., $t = \{([0], 0), -\}(COUNT \wedge ISEMPTY)$). Next, we build a negated path only consisting of the feature edges of the counterexample. Therefore, the path would consist of the edges $(3) \xrightarrow{COUNT} (4)$ and $(6) \xrightarrow{ISEMPTY} (7)$. The edge $(7) \xrightarrow{!x[0] <= 1} (9)$ will be ignored for the negated path. Next, we start the ARG construction again with the negated path (see Fig. 5.8). Since we only include feature edges, the test goal is only reachable on another configuration (i.e., state 9 in Fig. 5.8 is unreachable due to the negated path). For state $9'$ we now compute the presence condition as before (i.e., $!(COUNT \wedge ISEMPTY)$) and compute a test case (e.g., $t = \{([0], 0), -\}(!COUNT \wedge ISEMPTY)$). Lastly, we can restart the ARG construction with two negated paths. However, no other path will remain available to detect this test goal. Therefore, we created test cases for all products on which the test goal is reachable.

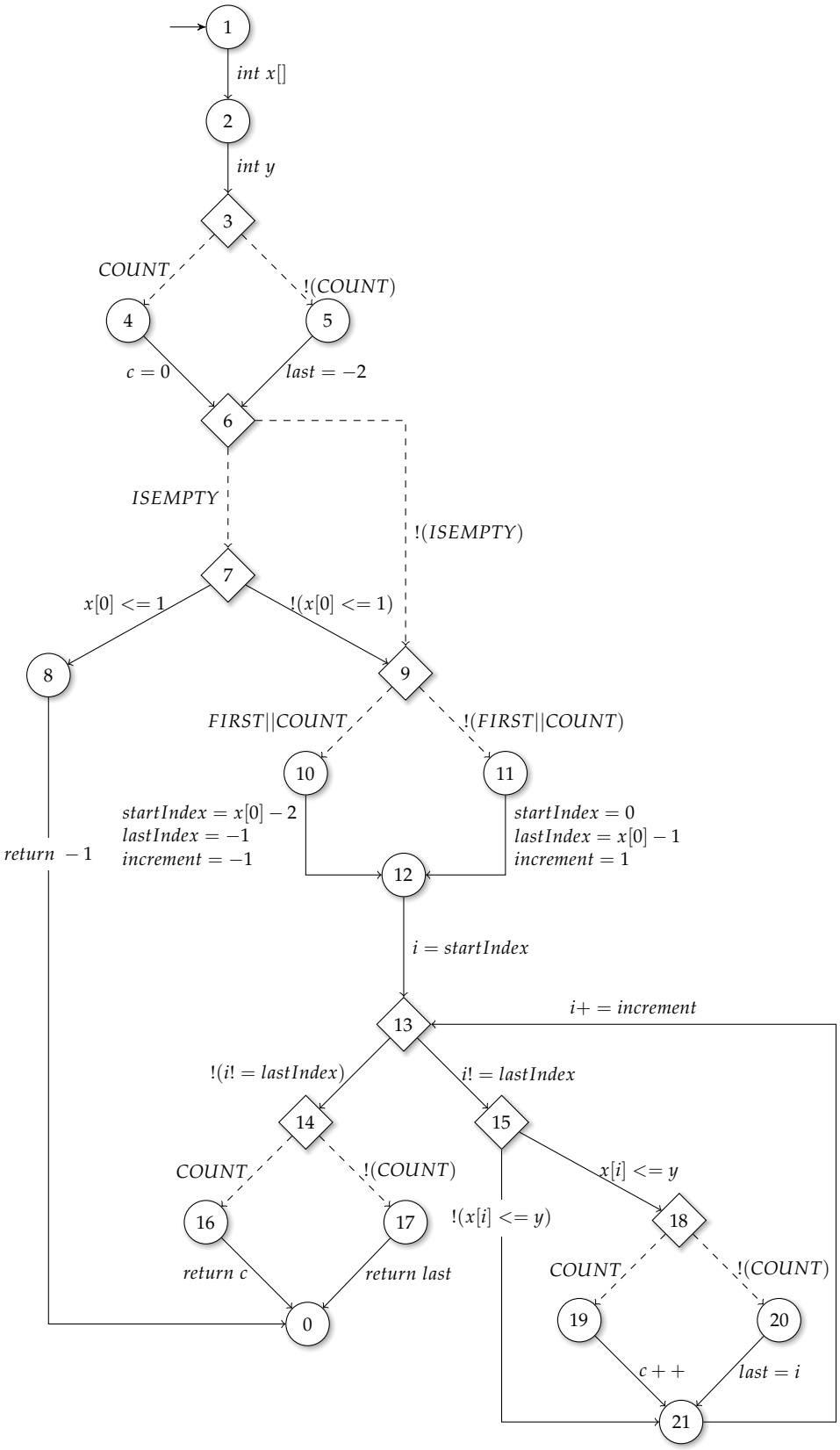


Figure 5.4: CFA SPL Implementation of the C-Function FIND_CONFIGURABLE

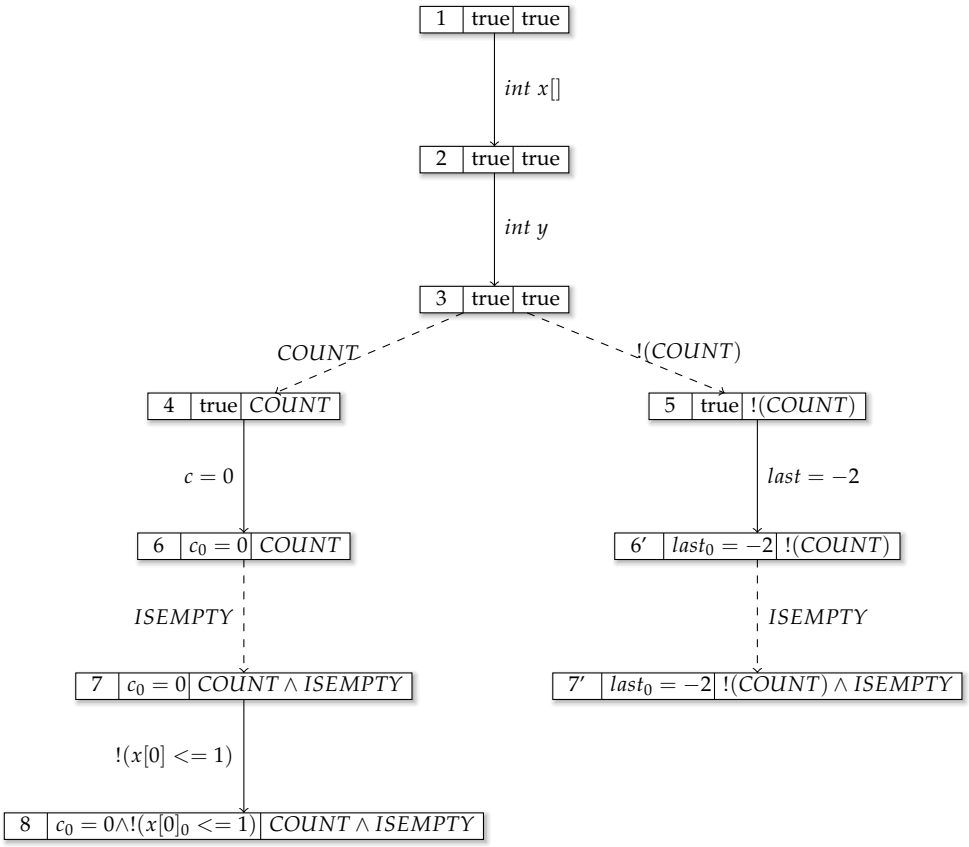


Figure 5.5: ARG of the SPL Implementation of the C-Function FIND_CONFIGURABLE

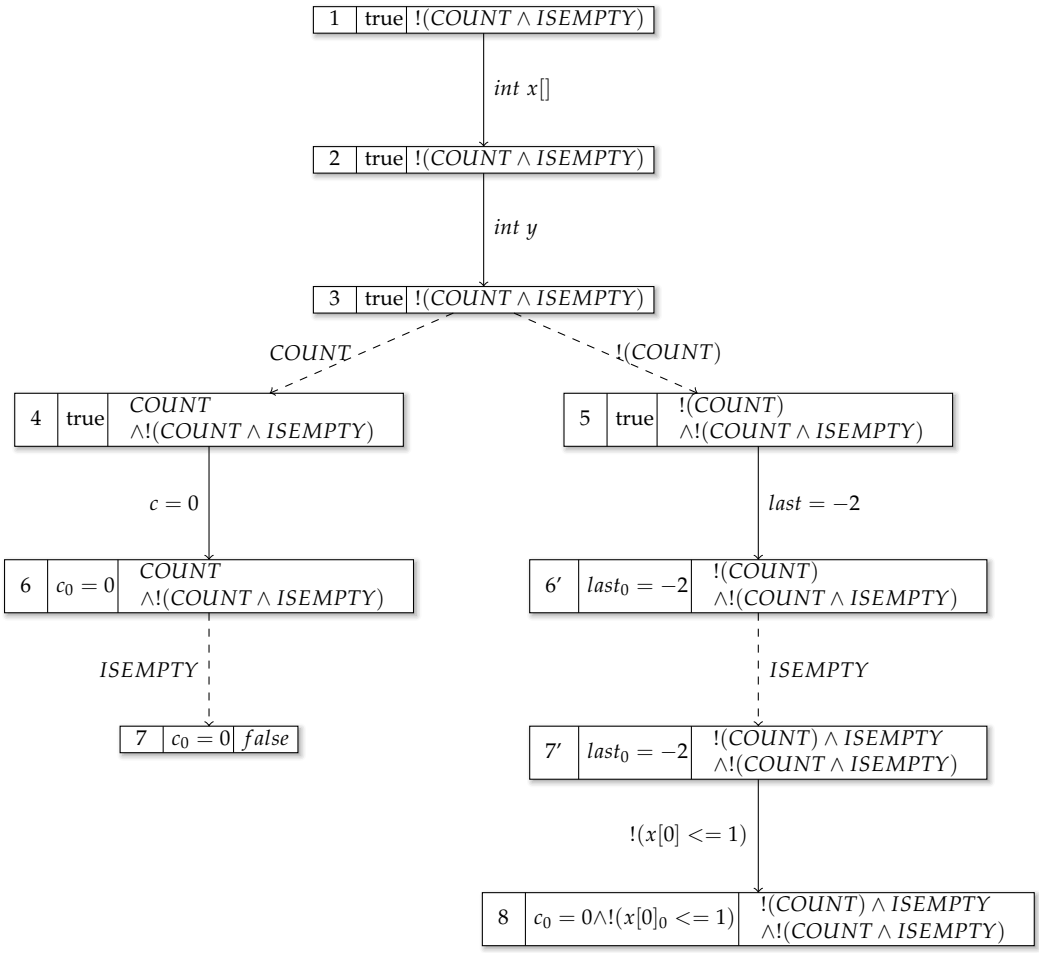


Figure 5.6: ARG of the Second Reachability Analysis of the C-Function FIND_CONFIGURABLE

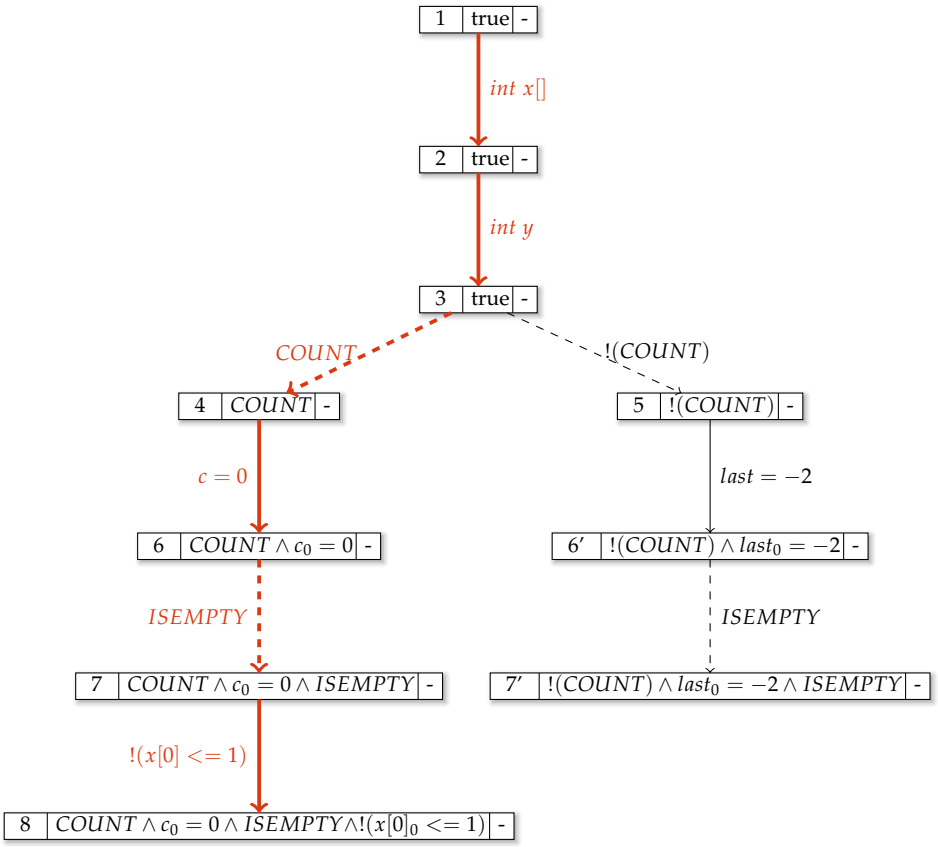


Figure 5.7: ARG with Negated Paths of the C-Function FIND_CONFIGURABLE

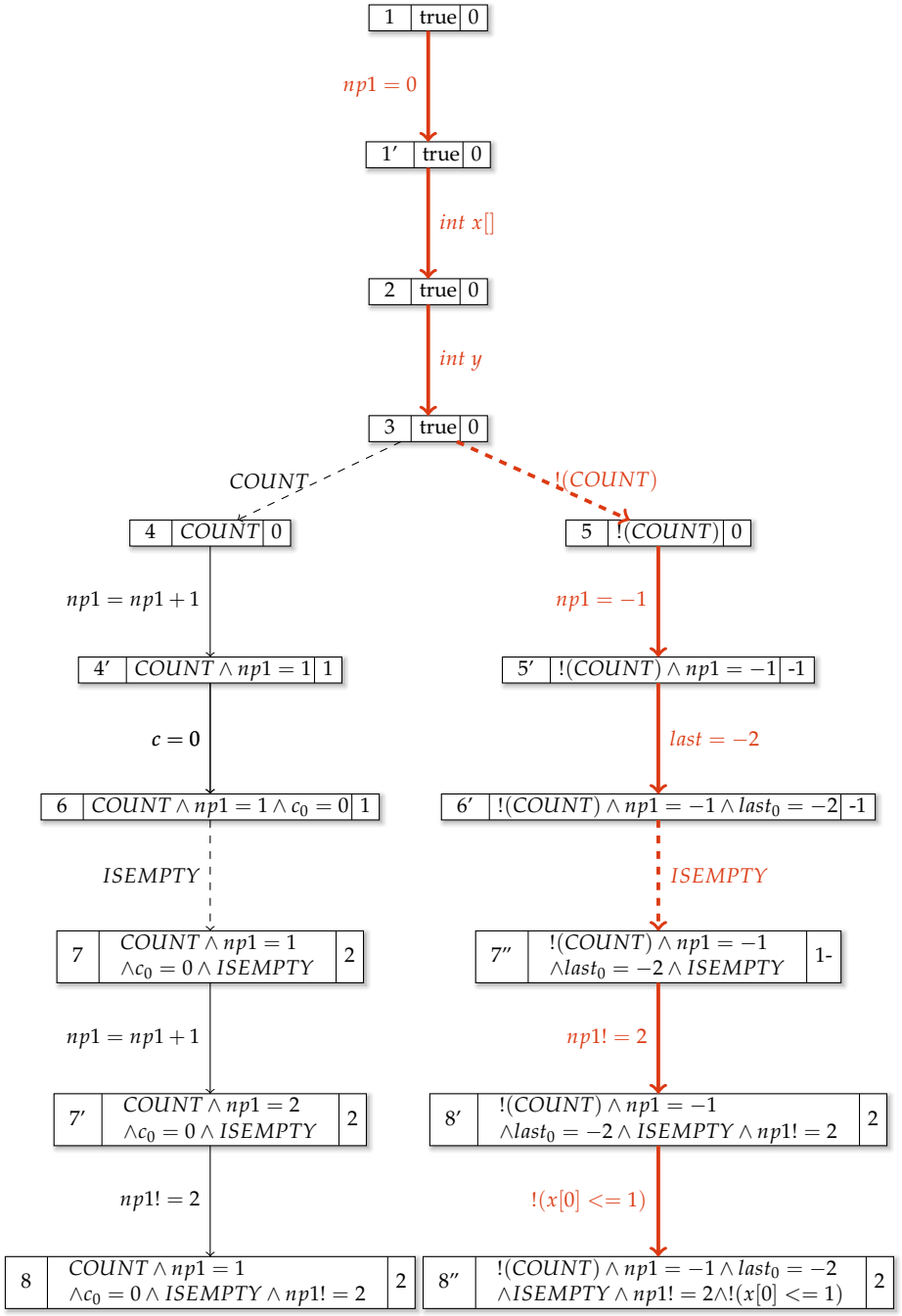


Figure 5.8: ARG of the Second Reachability Analysis with Negated Paths of the C-Function FIND_CONFIGURABLE

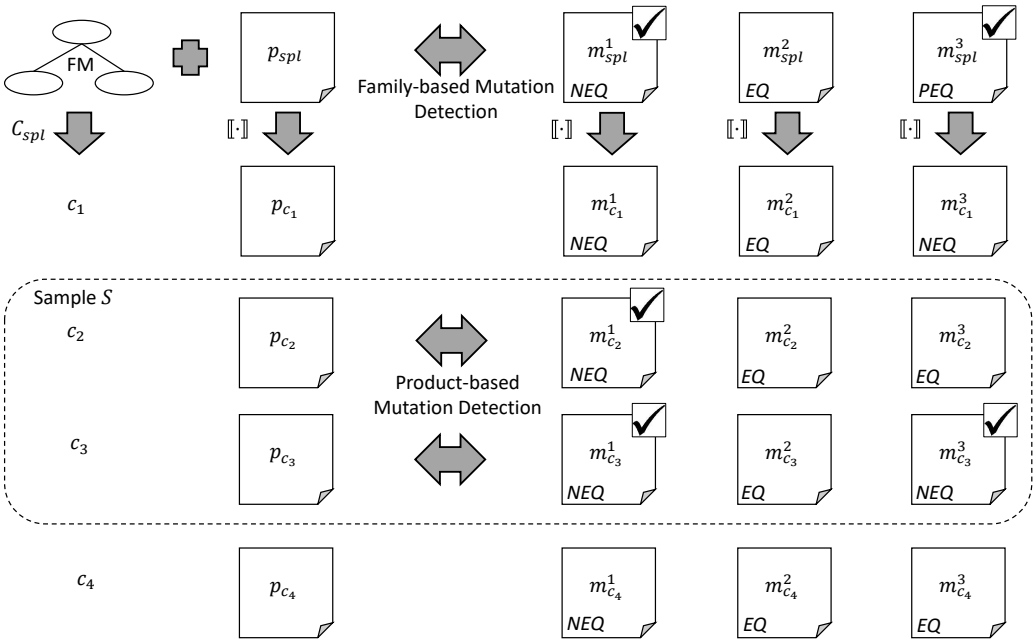


Figure 5.9: SPL Mutation-Testing Framework [155]

Note that both techniques naturally support consideration of feature models by encoding the feature model in C code, as described in Sect. 2.5

5.4 Product-Line Mutation Testing

As shown in the previous sections, the effectiveness of sampling strategies is often hard to measure. To this end, we developed a novel methodology based on family-based test-case generation and mutation testing to measure the effectiveness of sample-based software-product-line testing in an automated fashion.

5.4.1 *Overview*

Figure 5.9 (copied from [155]) shows the conceptional overview of our framework for measuring the effectiveness of sample-based testing.

Our methodology uses as input an SPL implementation p_{spl} and a feature model. Each valid configuration (e.g., $C_{spl} = \{c_1, c_2, c_3, c_4\}$ in Fig. 5.9) derived from the feature model corresponds to a single valid product (e.g., p_{c_1}, p_{c_2}, p_{c_3} and p_{c_4} in Fig. 5.9).

Next, we apply mutation operators tailored for SPLs (explained in the next section) to p_{spl}, resulting in a set of SPL mutants (see $M_{spl} = \{m^1_{spl}, m^2_{spl}, m^3_{spl}\}$ in Fig. 5.9).

Next, we use family-based test-case generation to determine, whether a mutant is semantically equivalent to the original SPL. This is possible, because we use software-model checking for test-case generation. If no test case exists that shows different behavior from the original SPL compared to the mutated SPL,

both are semantically equivalent. A detectable mutant is marked by the tick mark in Fig. 5.9. We consider three different kinds of SPL mutants.

- An SPL mutant is *non-equivalent* (*NEQ*) if every configuration derived from the mutated implementation is non-equivalent (e.g., m^1_{spl} in Fig. 5.9). This can be calculated by combining all presence conditions of the generated test cases as disjunction. If the resulting presence condition (see Sect. 5.3) is equal to the feature model, the mutant is *NEQ*. Hence, every non-empty sample may detect non-equivalent mutants.

- An SPL mutant is *equivalent* (*EQ*) if every program configuration derived from the mutated implementation is equivalent (e.g., m^2_{spl} in Fig. 5.9). If no test-case can be generated that detects the mutant, the mutant is *EQ*. Hence, no sample is able to detect equivalent mutants.

- An SPL mutant is *partially-equivalent* (*PEQ*) if it is neither *NEQ* nor *EQ* (e.g., m^3_{spl} in Fig. 5.9). This can be calculated as well by combining the presence conditions of the test cases as disjunction. If the resulting presence condition is not equal to the feature mode, the mutant is *PEQ*. Hence, a sample may detect a partially-equivalent mutant if it contains at least one non-equivalent test configuration.

For a given sample, we are now able to determine whether a product configuration is present, which would be able to detect a mutant (given a corresponding test case). For example, the sample $S = \{c_2, c_3\}$ in Fig. 5.9 would be able to detect m^1_{spl} and m^2_{spl} and, therefore, detect all detactable mutants.

Since only SPL mutants being *PEQ* are only detectable for certain products, mutants being *EQ* and *NEQ* are negligible for measuring the effectiveness of samples (as each sample with at least one product has equal effectiveness for *EQ* and *NEQ* mutants). For instance, if sample S in Fig. 5.9 contains c_4 instead of c_3, then S would fail to detect mutant m^3_{spl}.

In the following, we will describe our framework in more detail. First, we will explain the mutation operators used to mutate the SPL. Next, we will describe how we use test-case generation for SPLs to detect mutants and, lastly, how we measure the effectiveness of the samples.

5.4.2 *Product-Line Mutation Operators*

First, we describe the collection of mutation operators for SPLs (implemented in C code) used in this thesis. The operators are categorized in operators for common C code and operators specific for SPLs. In the following, we will refer to an SPL mutant resulting from applying any mutation operator to p_{spl} as m_{spl}. Additionally, we denote a set of SPL mutants as M_{spl}.

The overview of the mutation operators used is given in Tab. 5.1 (adapted from [155]). These mutation operators will be explained in the following.

Code-Mutation Operators. To mutate C code, we use established mutation operators mutating conditional expressions and assignment statements. [103,

Table 5.1: Overview Mutation Operators Applicable for Mutating SPL Mappings (adapted from [155])

		Condition	Assignment	Presence Condition
Replacement	Constant /Variable	x	x	x
	Arithmetic Op.	x	x	
	Logic Op.	x	x	x
	Relation Op.	x	x	
	In-/Decrement	x	x	
Insertion	Absolute Value	x	x	
	In-/Decrement	x	x	
Other	Delete			x
	Negate	x		x

11, 3] To this end, our framework incorporates 77 C code mutation operators as defined by Agrawal et al. [3]. The mutation operators are implemented in the C code mutation tool MiLu [102], which we utilize in our framework. The C Code mutation operators include different mutations, such as replacing memory allocation calls (e.g., replacing occurrences of `malloc` by `alloca`), replacing logic-, arithmetic-, and relational-operators (e.g., replacing occurrences of `+` by `/`), and insertions of pre-/postfix in-/decrements (e.g., changing occurrences of variables `i` to `i++` or `-i`).

Since feature variables need special mutation operators (e.g., replacing a feature variable by a standard variable might break the SPL), we exclude feature variables from C code mutation operators. Furthermore, our framework incorporates additional mutation operators tailored for SPLs by handling feature-based variability encoding within presence conditions.

Feature-Mapping-Mutation Operators. Additional mutation operators tailored for SPLs where developed at the TU Braunschweig [155]. To this end, existing mutation operators were adopted to mutate presence conditions. In total, we used five different SPL mutation operators. Those operators mimic different kinds of variability errors in terms of erroneous feature-mappings onto conditional code.

Table 5.1 provides an overview on the different categories of mutation operators included in our framework, grouped into *replacement*, *insertion*, and *other*.

- **Feature Replacement:** This operator replaces an occurrence of a feature in a presence condition by another feature variable defined in the feature model (e.g., replacing the feature *FIRST* with *LAST* in line 17 in our running example in Fig. 2.13).
- **Logical Operator Replacement:** This operator replaces an occurrence of a binary logical operator in a presence condition by another binary logical operator (e.g., replacing || with && in line 17 in our running example in

Fig. 2.13). Currently, we limit our mutation operator to only replace || with && and vice versa.

- **Feature Deletion:** This operator replaces an occurrence of a feature variable from a presence condition by replacing the variable by either 1 (i.e., *true*) or 0 (i.e., *false*) (e.g., replacing the feature $FIRST$ with 0 in line 17 in our running example in Fig. 2.13).

- **Presence Condition Negation, Feature Negation:** Those two operators negate a presence condition of a feature by inserting or deleting occurrences of negation operators (i.e., !) in presence conditions (e.g., inserting a negation operator for the presence condition in line 17 in our running example in Fig. 2.13 results in $\#if\ !(defined(FIRST)||defined(COUNT))$, inserting a negation operator for the feature $FIRST$ would result in $\#if\ !defined(FIRST)||defined(COUNT)$).

In case any of those feature mutations leads to a presence condition which cannot be fulfilled in combination with the feature model, the source code corresponding to the presence condition is unreachable code (and, therefore, highly likely to be a detectable mutant).

5.4.3 *Family-Based Product-Line Mutant Detection*

In Sect. 4.3.1, we explained how to create programs encoding the detection of a bug or mutant as a reachability problem for test-case generation. Based on the mutation operators for SPLs, we now lift the notion of standard mutation detection to mutation detection for SPLs using the mutation operators explained before. We denote C as the set of valid configurations for the software product line and $c \in C$ as a single valid configuration. Additionally, we denote P_{spl} as the code base for the software-product line and p_c as the product derived using configuration c. Next, we denote the set of mutants of P_{spl} as $M_{spl} = \{M_1, M_2, M_3, ...\}$ and $m_{i,c}$ as the mutant of M_i derived by using configuration c.

A mutant M_i is detectable if at least one configuration $c \in C$ exists such that the derived programs p_c and $m_{i,c}$ enables the detection of the mutant.

For an SPL mutant M_i to be detectable a configuration $c \in C$ must exist that $p_c \neq m_{i,c}$ (i.e., p_c is syntactically different from $m_{i,c}$, therefore, the mutant is present in the derived product). Additionally, there must exist at least one test case t which leads to different output behavior when executed on p_c compared to $m_{i,c}$ (i.e., $p_c(t) \neq m_{i,c}(t)$). If a test case $t \in T_c$ for the given configuration exists with $p_c(t) \neq m_{i,c}(t)$, the test suite T_c is able to detect the mutant.

Therefore, we can apply the notion of RIPR criteria for mutation testing of single programs to SPL mutation testing with respect to a given configuration c. [103, 11]

- **Reachability (R):** The mutated program location in $m_{i,c}$ is reachable by at least one test case $t \in T_c$.

- **Infection (I):** Reaching the mutated program location in $m_{i,c}$ leads to an modified program state compared to p_c.

- **Propagation (P):** The modified program state propagates to further program states.
- **Reveal (R):** The propagated modified program state leads to an observable modified output of $m_{i,c}$ compared to the output of p_c.

Therefore, a mutant $M_i \in M_{spl}$ is an *equivalent SPL mutant* if no product configuration exists for which the RIPR criteria holds (i.e., for all configurations $c \in C$ either $p_c = m_{i,c}$ or no test case exists for that $p_c(t) \neq m_{i,c}(t)$ holds). Additionally, an SPL mutant is *partially equivalent* if there exists at least one configuration $c \in C$ for that the mutant is detectable, but there also exists at least one configuration for that the mutant is not detectable. Lastly, we call a mutant *non-equivalent* if it is detectable for all configurations.

We now have the means to measure the effectiveness of samples based on mutation detection in a product-by-product manner. However, deriving the mutant $m_{i,c}$ of each configuration $c \in C$ and check whether the mutant is detectable for c (and, therefore, identifying the subset $C_{M_i} \subseteq C$ for that the mutant is detectable) is infeasible due to the reasons mentioned before. Therefore, we use the test-case generation technique described in Sect. 5.3 combined with the technique to generate modification-revealing test cases described in Sect. 4.3.1. This enables us to create a test-suite that contains a test case for each configuration on which the mutant is detectable. Using those techniques we are now able to compute the effectiveness of a sample $S \in C$ for a given set of n mutants M_{spl} as:

$$\mathit{effectiveness}(M_i, S) = \begin{cases} 1, & S \cap C_{M_i} \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$

$$\mathit{effectiveness}(S) = \frac{\sum_{i=1}^{n} \text{effectiveness}(M_i, S)}{n} \tag{5.2}$$

5.5 Implementation

As explained in Sect. 5.3, we utilize the negated paths from Sect. 4.3.2. Therefore, the implementation stays similar with only minor adjustments. Figure 5.10 shows the modules used (with normal border) and extended (with thick border) for test-case generation for SPLs.

Tiger MultiGoal Algorithm. The Tiger MultiGoal Algorithm is modified to support test-case generation for SPLs by two implementation artifacts. First, when creating a negated path from a counterexample, only the feature edges are used instead of all branching edges. Second, for generating test cases, the feature variables are used as presence condition and ignored as input parameters. Additionally, there exist a new configuration to determine feature edges by specifying a common prefix (e.g., for our running example, we would rename the features to *FEAT_FIRST*, *FEAT_LAST*, *FEAT_COUNT* and *FEAT_ISEMPTY* and specify *FEAT_* as prefix).

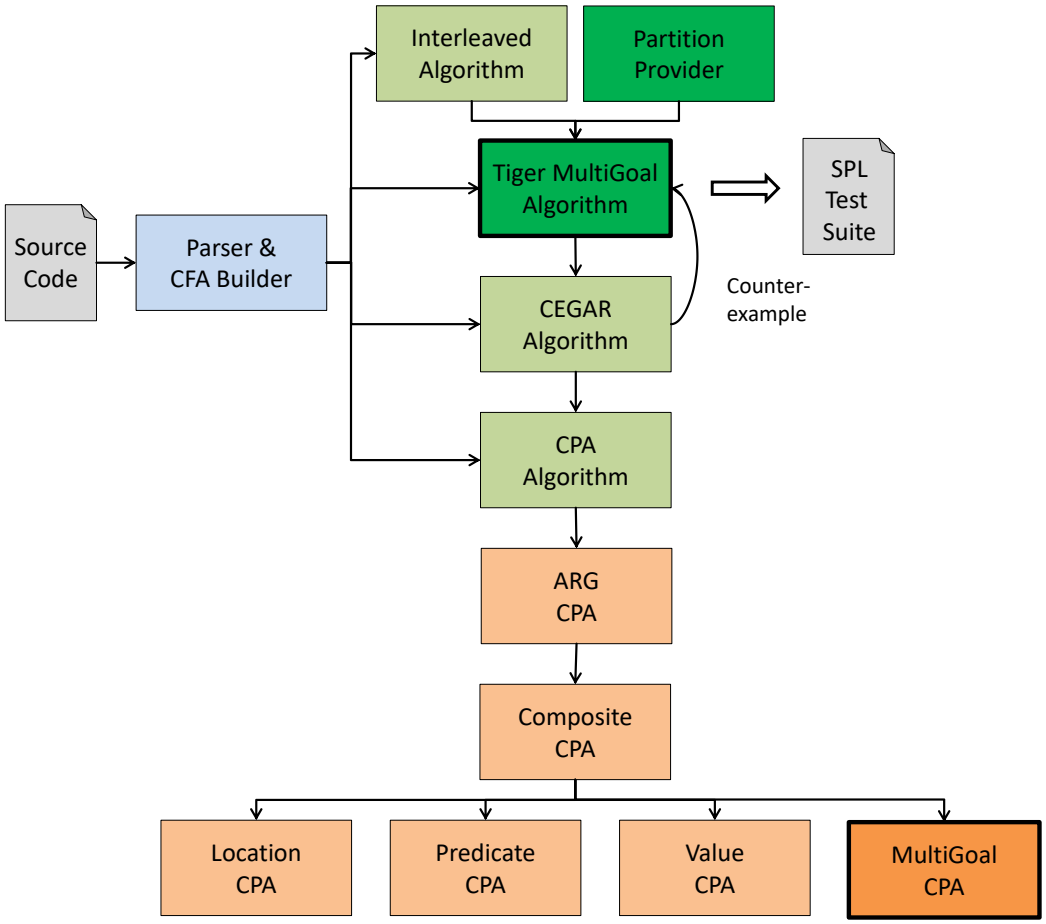


Figure 5.10: Modules for Test-Case Generation for Software-Product Lines with CPAchecker (adapted from [21])

MultiGoal CPA. The MultiGoal CPA is modified to only take feature edges into account for negated paths when computing a successor state.

The modifications to MultiGoal CPA and Tiger MultiGoal Algorithm are enabled or disabled by a common configuration option. This is necessary since negated paths for SPLs cannot be combined with negated paths for multi-test-case generation in the current implementation.

5.6 Evaluation

The framework shown in this chapter enables us to measure the effectiveness of different kinds of SPL analysis strategies in general, and sample-based SPL testing in particular.

In our evaluation, we consider SPLs implemented in C. Additionally, the SPL implementation needs to be lifted to run-time variability (see Sect. 5.3), as well as have their feature model encoded in C code (see Sect. 2.5.3). To measure effectiveness and $\mathit{efficiency}_{samplesize}$, we use real-world SPL projects, as well as synthetically created SPLs.

To this end, we first derive a set of mutants from the SPL implementation (see Sect. 5.4) using standard mutation operators and SPL specific mapping-mutation operators (see Sect. 5.4).

We measure the effectiveness of a given sample by measuring how many of the generated mutants are potentially detectable by at least one configuration of the sample (see Sect. 5.4). As shown in Sect. 5.3, we generate a software-product-line test-suite containing test cases detecting the mutant for each product on which the mutant is detectable (at least if the test-case generation was successfully terminated without running into a timeout). The test cases contain information (encoded as presence condition over feature variables) for which configuration the test case is valid. This information can now be used to determine if a given sample contains a configuration for that one of the test cases is valid. If the sample does contain such a configuration, the sample is potentially able to detect the mutant.

In this section, we first describe our research question and experimental setup. Next, we show how we synthetically construct SPLs from single products to further investigate the impact of different SPLs and feature models. Lastly, we show and discuss our evaluation results and the threats to validity.

5.6.1 *Research Questions*

We consider the following research questions.

(RQ1) How does the value of τ impact effectiveness of τ-wise sample-based PL testing strategies?

- **(RQ1.1)** Does increasing $\tau = 2$ to $\tau = 3$ in τ-wise sampling lead to a significant improvement of effectiveness, as compared to increasing $\tau = 1$ to $\tau = 2$?
- **(RQ1.2)** Does decreasing $\tau = 2$ to $\tau = 1$ in τ-wise sampling lead to a significant improvement of $\mathit{efficiency}_{\mathit{samplesize}}$, as compared to decreasing $\tau = 3$ to $\tau = 2$?
- **(RQ1.3)** Does 2-wise sampling constitute the best trade-off between effectiveness and $\mathit{efficiency}_{\mathit{samplesize}}$, as compared to 1-wise sampling and 3-wise sampling?

(RQ2) What is the impact of partially-equivalent mutants in measuring τ-wise sample-based PL testing strategies?

- **(RQ2.1)** How many partially-equivalent mutants occur on average, as compared to the total number of mutants?
- **(RQ2.2)** To what extent does the detection of a mutant depend on selection/deselection of particular sets of features?
- **(RQ2.3)** How many different test configurations are capable on average to detect a partially-equivalent mutant?

5.6.2 *Experimental Setup*

Next, we will describe the evaluation methods and experimental design used in our experiments to address the research questions.

For practical reasons, we limit our consideration of τ to $\tau \in 1,2,3$, which are the most relevant sampling criteria in practice.

Note that for the evaluation, we used the old implementation for test-case generation for SPLs with the blocking-clauses explained in Sect. 5.3. The reason is that the evaluation was executed before the development of the new test-case generation with negated paths was complete. However, the results should not differ for the new implementation, as negated paths should only affect the $\mathit{efficiency}_{CPU}$ of test-case generation, which is not taken into account for this evaluation.

Subject Systems. We consider preprocessed C programs as our subject systems, which consist of an entry-function (i.e., the function under test) and the variability encoded as runtime variability. Our selection of subject systems consists of the following real-world SPLs.

- **VIM**: VIM (v8.1)[1] is a highly-configurable test editor and a new implementation of the editor VI. Many classes implemented in VIM are highly-configurable by a variety of different features.
- **BusyBox**: BusyBox (v1.24.2)[2] is a reimplementation of standard Unix tools written in C for systems with limited resources. It contains multiple tiny versions of common UNIX utilities within a single executable. Many tools in BusyBox are implemented as configurable SPLs (i.e., families of program variants).

As our family-based mutation-detection technique is based on the generation of mutation-detecting test cases at unit level, we consider in our experiments a suitable selection of functions extracted from the respective subject systems. In particular, we selected only those functions depending on at least four features (to potentially provide sufficient feature interaction) and having a return type other than `void`.

Synthetic Software-Product Lines. Additionally to the real-world SPLs used as subject systems, we also use **synthetically** generated software-product-line implementations in our experiments. This allows us to investigate the impact of different product-line characteristics (different feature models, etc.) in more detail. To this end, we utilize the tool SiMPOSE[3], which allows for the generation of SPLs based on multiple products (i.e., it merges several product implementations to an SPL). SiMPOSE takes as input an arbitrary number N of program variants and applies N-way merging to generate an SPL implementation incorporating all program variants. SiMPOSE uses superimposition of control-flow representation of the products to generate the SPL. To generate multiple program variants from

[1] https://www.vim.org/vim-8.1-released.php
[2] https://busybox.net/
[3] http://pi.informatik.uni-siegen.de/Projekte/variance/tosem18/

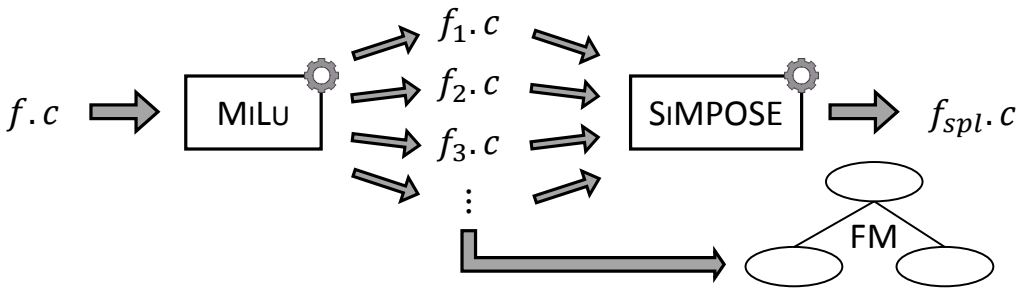


Figure 5.11: Synthetic SPL Generation [155]

Table 5.2: Functions Extracted from the Subject Systems [155]

Function	Subject System	Synthetic	#F	LOC
01: calculate	—	✓	5	78
02: is_method_1_triggered	BusyBox	✓	5	86
03: lcm	—	✓	5	105
04: rand_int	—	✓	5	49
05: passwd_main	BusyBox	✗	4	84
06: deluser_main	BusyBox	✗	6	476
07: minimized_get_varp	VIM	✗	7	40
08: gui_get_base_height	VIM	✗	7	35
09: gui_init_check	VIM	✗	9	107
10: insecure_flag	VIM	✗	6	45

single products, we utilize the tool MiLu by introducing mutants into the existing program. Each mutant is considered a new program variant.

The overall procedure is shown in Fig. 5.11 and consists of the following steps.

1. We take an existing C function $f.c$ as input.
2. We apply the tool MiLu to $f.c$ to generate a set of (potentially equivalent) mutants $f_1.c, f_2.c, \ldots$ from $f.c$.
3. We apply SiMPOSE to create an SPL $f_{spl}.c$ from the set of mutants.
4. We create a feature model for $f_{spl}.c$ where the difference between the original function $f.c$ and a mutant $f_1.c$ correspond to a feature. The feature model has a feature for each mutated line of code within an or-group directly after the root feature. Each mutant modifying the same line of code is put into an alternative group directly after the feature for the corresponding line of code. This ensures, no line of code is included twice in the same product due to superimposition. We further annotate the feature model with randomly generated cross-tree constraints.

The functions extracted from our subject systems that are used for our evaluation are listed in Tab. 5.2. Additionally, the table shows the number of features $\#F$ and lines of code LOC of the extracted functions.

Data Collection. As described before, we first introduce mutants to our subject systems and create test-suites using test-case generation for SPLs (as explained in Sect. 5.3). However, as we are concerned with the evaluation of sampling strategies, measurement of test-case generation is for our evaluation out of scope.

Next, we generate different τ-wise samples and measure their effectiveness and $efficiency_{samplesize}$ to answer research question **RQ1.1**, **RQ1.2** and **RQ1.3**. We limited our consideration of τ to $\tau \in 1, 2, 3$ for practical reasons. We calculated the effectiveness of samples based on Formula 5.2 in Sect. 5.4.3.

We calculated the efficiency of a sample in terms of the number of possible configurations as follows:

$$efficiency_{samplesize}(S) = 1 - \frac{|S|}{|C|} \tag{5.3}$$

To evaluate the trade-off, we calculated the average values of effectiveness and $efficiency_{samplesize}$ and compared both. To measure the trade-off, we used the following equation.

$$trade\text{-}off(S) = effectiveness(S) * efficiency_{samplesize}(S) \tag{5.4}$$

To answer **RQ2**, we generated 30, 50, and 100 mutants randomly. We limited our evaluation to 100 mutants, as for several of our subject systems, no more than 100 different mutants could be generated. To answer **RQ2.1**, we measured the number of partially-equivalent mutants and compared them to the total amount of mutants. To answer research question **RQ2.2**, we measured for each mutant the number of features that must be selected or deselected in order to detect the mutant (i.e., the number of features responsible for detecting the mutant). Lastly, to answer **RQ2.3**, we measured for each mutant the number of configurations being able to detect the mutant and calculated the average number of configurations being able to detect mutants of each subject system.

Measurement Setup. All tools used throughout the evaluation were executed on a Windows 10 machine with an Intel Core i7-7500 CPU with a 2.7-3.5GHz clock rate and 16GB of RAM. We limited the CPU time of the detection of a single mutant to 600 seconds for practical reasons. We did not limit the CPU time during sample generation of mutant generation, as the CPU time for the sample generation was insignificant. For generating mutants with MiLu, we used the Linux subsystem on Windows 10. The mutant detection, sample generation, and evaluation were executed with Java 1.8.0_161 and limited the Java heap to 6GB.

5.6.3 *Results*

Fig. 5.15 shows the results of our experimental evaluation concerning research question **RQ1**. The results of our evaluation regarding **RQ2** are shown in Fig. 5.16.

RQ1.1 Effectiveness. Fig. 5.12 shows the effectiveness of τ-wise sampling for our subject systems. The figure shows that generating τ-wise sampling with

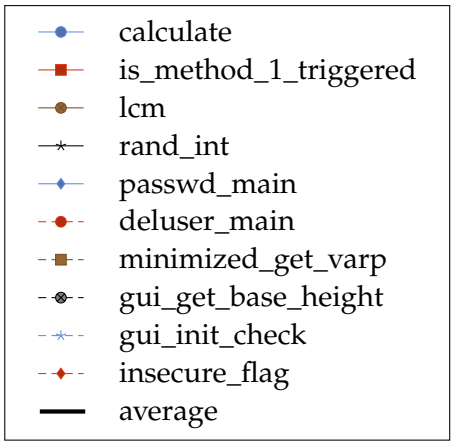


Table 5.3: Legend for Fig. 5.12 and Fig. 5.13

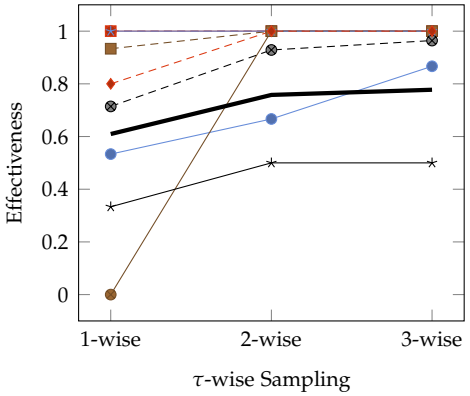


Figure 5.12: Effectiveness of Samples

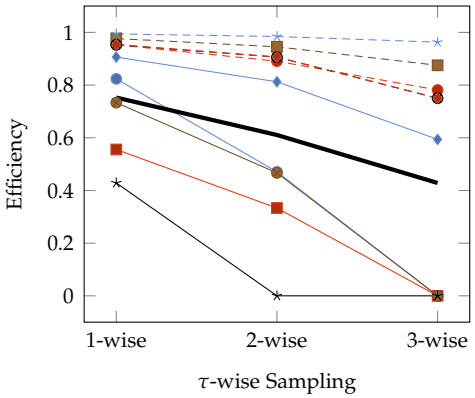


Figure 5.13: $efficiency_{samplesize}$ of Samples

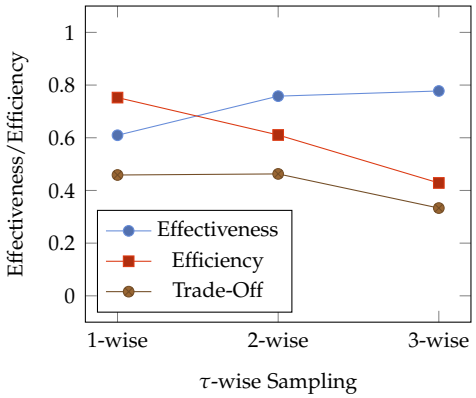
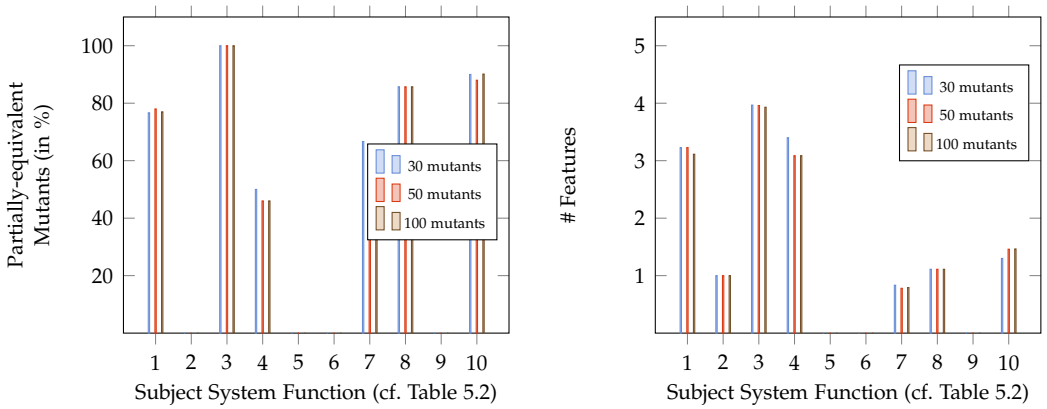


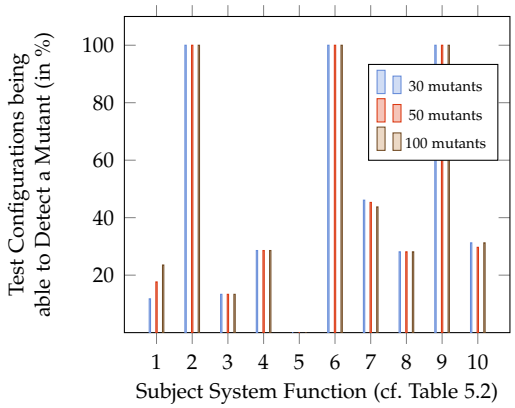
Figure 5.14: Trade-off

Figure 5.15: Results for **RQ1.1**, **RQ1.2** and **RQ1.3** [155]



(a) Percentage of Partially-equivalent Mutants

(b) # Features Necessary to Detect a Mutant



(c) Percentage of Test Configurations being able to Detect a Mutant

Figure 5.16: Results for **RQ2.1**, **RQ2.2** and **RQ2.3** [155]

$\tau = 2$ leads to significantly higher effectiveness compared to $\tau = 1$. However, for most subject systems stepping from $\tau = 2$ to $\tau = 3$ only increases the effectiveness by either a much smaller margin or did not increase effectiveness at all.

RQ1.2 Efficiency. Fig. 5.13 shows the $efficiency_{samplesize}$ of τ-wise sampling for our subject systems. The figure shows that stepping from $\tau = 1$ to $\tau = 2$, the $efficiency_{samplesize}$ decreases for all subject systems. Stepping from $\tau = 2$ to $\tau = 3$ decreases $efficiency_{samplesize}$ by an even larger margin for most subject systems.

RQ1.3 Trade-Off. Figure 5.14 shows the average values of both effectiveness and $efficiency_{samplesize}$ for our subject systems, and the trade-off (see equation 5.4).

RQ2.1 Partially-Equivalent Mutants. Figure 5.16a shows the average amount of partially-equivalent mutants compared to the total number of mutants (in percentage). The figure shows that for 30, 50, and 100 mutants generated, the results are similar. However, the number of partially-equivalent mutants differs between different subject systems.

RQ2.2 Feature Dependencies. Lastly, Fig. 5.16b shows the number of features that have to be selected or deselected to detect a partially-equivalent mutant. The number of features ranges from 0 to 3.96 for our subject systems.

RQ2.3 Test Configurations per Mutant. The average number of test configurations able to detect a mutant compared to the total number of valid configurations for each subject system is depicted in Fig. 5.16c. As for **RQ2.1**, we observe similar results for 30, 50, and 100 mutants generated.

5.6.4 *Discussion and Summary*

RQ1.1 Effectiveness For most τ-wise samples generated for our subject system, the effectiveness increased when stepping from $\tau = 2$ to $\tau = 3$, however, by a much smaller margin compared to stepping from $\tau = 1$ to $\tau = 2$. However, there are some exceptions. The mutants generated for the functions *is_method1_triggered*, *deluser_main* and *gui_init_check* are all non-equivalent and, therefore, detectable by all samples (if at least one configuration is present). This happens, if only core functionality is mutated and the detection of the mutant is not prevented by any feature. The mutants generated for function *passwd_main* were all equivalent and, therefore, not detectable on any configuration. This happens, if the mutant has no impact on the return value. As a consequence, the effectiveness for all samples for these subject systems is always 1 or 0. Therefore, increasing the sample size by increasing τ does not lead to higher effectiveness.

Another exception is the function *calculate*. For the mutants generated for this function the increase of effectiveness from $\tau = 2$ to $\tau = 3$ is higher compared to the increase of effectiveness from $\tau = 1$ to $\tau = 2$. The reason for this phenomenon is that for most mutants generated for this function, the number of features necessary to be selected or deselected is more than 2 (and, therefore, the mutants are not guaranteed to be detected by $\tau = 2$).

RQ1.2 Efficiency For most subject systems, the $efficiency_{samplesize}$ of samples decreased by a larger margin when stepping from $\tau = 2$ to $\tau = 3$ compared to stepping from $\tau = 1$ to $\tau = 2$. The only exception is caused by function *rand_int* because 2-wise samples already contain all valid product configurations and, therefore, has an $efficiency_{samplesize}$ of 0. To conclude, the decrease of $efficiency_{samplesize}$ from $\tau = 2$ to $\tau = 3$ is, on average, much higher than from $\tau = 1$ to $\tau = 2$.

RQ1.3 Trade-Off Fig. 5.14 shows that when stepping from $\tau = 1$ to $\tau = 2$ effectiveness increases by approx. 24.3% and $efficiency_{samplesize}$ decreases by approx 23.2% and, therefore, the trade-off increases. However, when stepping from $\tau = 2$ to $\tau = 3$ effectiveness increases only by approx. 2.6% while $efficiency_{samplesize}$ decreases by approx. 42.5% and, therefore, the trade-off decreases.

The results of our evaluation for **RQ1** confirm the current state of research in literature. This implies that our framework can be well used to evaluate further sampling strategies.

RQ2.1 Partially-Equivalent Mutants. For most of our subject systems, some of the generated mutants are partially-equivalent. For those subject systems, the amount of partially-equivalent mutants is between 40% to 80%. However, the mutants generated for the functions *passwd_main*, *is_method1_triggered*, *deluser_main*, and *gui_init_check* are all either non-equivalent or equivalent. This might be due to two reasons. First, the mutation operators used for those systems might not be the correct choice. Second, the functions might have few or no mutatable parts of code affecting the return values.

RQ2.2 Feature Dependencies. To measure the feature dependencies to detect mutants, we measured the number of features that have to be either selected or deselected to detect a given mutant. For partially-equivalent mutants of our subject systems, between 0,78 and 3,96 features are needed to either select or deselect to detect the mutant (i.e., ignoring features for which the mutant can be detected no matter the selection). Therefore, to detect all detectable mutants the choice of τ is highly depending on the subject system. This is consistent with the usual assumptions in literature and, therefore, reinforces our results.

RQ2.3 Test Configurations per Mutant. The mutants generated for the functions *is_method1_triggered*, *deluser_main*, and *gui_init_check* were all non-equivalent. Therefore, every possible product configuration is able to detect these mutants. The mutants for the function *passwd_main* were all equivalent, leading to 0 configurations being able to detect the mutant. For the remainder of the functions, between 11,76% and 46,09% of the configurations are able to detect a mutant. This result is also consistent with the assumptions about the detectability (i.e., most bugs are found with few feature interactions) of bugs in literature.

Additionally, the saturation of our results for **RQ2.1-2.3** is (mostly) reached by 30 mutants (i.e., 50 and 100 mutants lead to the same results). This might increase $efficiency_{samplesize}$ of our framework since we can rely on fewer bugs for future measurements.

5.6.5 *Threats to Validity*

Internal Validity. The selection of our mutation operators and the application of those operators might pose a threat to our internal validity. However, we selected mutation operators that lead to compilable mutants and separated mapping-mutating operators from code-operators to not mix features and program variables. Additionally, we randomly selected mutants for our evaluation to avoid possibly favorable mutation operators. Therefore, our selection of mutation operators should not harm the validity of our results. The full list of used mutation operators is shown in Appendix B.

Additionally, we limit our consideration of subject systems to C programs. However, our methodology should be applicable to other imperative programming languages (e.g., most object-oriented languages), too. For those languages, we expect similar results.

Another threat to validity is the limitation of our evaluation to unit testing. However, unit testing is one of the most established testing techniques in practice and, therefore, highly relevant. Nonetheless, our methodology can easily be used for integration or system testing, however might have issues with scalability.

Lastly, our framework is currently limited to boolean feature variables and corresponding presence conditions. However, our framework might be extended to also support extended feature models [107] to support non-boolean feature variables.

External Validity. The selection of our subject systems (including the synthetic SPLs) may constitute a threat to the external validity of our evaluation. The mutants generated for the functions from VIM behave similarly to the mutants of our synthetic SPLs. However, mutants generated for BusyBox are often non-equivalent. While the number of subject systems is small and the results slightly differ, the SPLs have a reasonable size and complexity, in our opinion. Additionally, the real-world subject systems used in our evaluation constitute a reliable reference.

Another threat to external validity may be the lack of comparison to other tools. However, to the best of our knowledge, no similar technique to use mutation testing for measuring the effectiveness of samples has been described so far.

Lastly, we rely on third-party software, namely CPAchecker, MiLu, and FeatureIDE. However, all three tools are established and have been used in other experiments in the past. Therefore, we expect them to produce sound results.

5.7 Related Work

In the following, we will discuss related work with a focus on the two most relevant aspects, namely measuring the effectiveness of samples and the application of mutation testing to SPLs.

For a broader overview on SPL testing, we refer to dedicated surveys [139, 68, 55, 115, 63]. For an overview on sample-based SPL testing, we also refer to dedicated surveys [4, 126, 174].

Usually, sample-based testing is evaluated in sampling efficiency, testing efficiency and effectiveness [174]. In this case, sampling efficiency corresponds to the CPU time needed to compute a sample, and testing efficiency refers to the CPU time needed to test such a sample. Additionally, the effectiveness is mainly evaluated as the number of covered feature interactions. [7, 69, 94, 104, 125, 127, 142, 73, 140, 96, 64, 65, 72, 18, 110].

While there exist multiple techniques that guarantee the coverage of feature interaction to a certain degree (e.g., τ-wise sampling), there might be even feature interaction of a higher degree covered within the sample. [76, 130, 142, 106, 7, 138, 52, 92, 105]

This effectiveness metric can easily be computed solely relying on the feature model. Therefore, it does not incorporate any properties of the SPL implementation (e.g., if there are no feature interactions within the implementation $\tau > 1$, makes no sense). Additionally, in contrast to our approach those works do not

take effectiveness in terms of bug detection (or mutant detection) into account which is not always corresponding to the feature coverage.

However, there exist work evaluating effectiveness on other or additional properties. Devroey et al. [58, 59] evaluated the effectiveness of a given sample in terms of feature and behavioral model coverage. Tartler et al. [170] also incorporate the implementation to evaluate effectiveness. To this end, they focus on SPLs with compile-time variability (i.e., compiler directives) and measure the coverage of `#ifdef` blocks of a given sample (i.e., if the code block is included in some configurations and, therefore, could be executed). This metric also includes feature-to-code mapping. However, the interactions caused by multiple `#ifdef` blocks are still out of scope. Again, those works do not take actual bug detection of the samples into account.

There also exist works that evaluate the effectiveness of samples in terms of real-fault coverage. [1, 82, 16, 133, 171, 73, 158, 159] The work of Sánchez et al. [158, 159] proposed the Drupal case study for measuring the effectiveness of samples based on real faults. Fischer et al. used the faults of the Drupal case study in combination with feature interaction coverage to evaluate the effectiveness of samples. [73] In 2014 Tartler et al. evaluated `#ifdef` block coverage paired with real faults. [171] Medeiros et al. [133] measured effectiveness based on faulty configurations in open-source C progrems (e.g., BusyBox and VIM). Lastly, Halin et al. [82] build test cases able to be executed on all variants of the industrial-size system JHipster and tested all products in a product-by-product manner. [82] The resulting faults discovered were next used as effectiveness measurement for sampling.

While real faults are (probably) the best source for evaluating effectiveness, their number is usually very low, and real faults are only known for few SPL implementations. For this reason, our approach is more general and allows the measurement of effectiveness for most SPL implementations (as long as they are technically supported).

To introduce synthetic faults, it is possible to apply mutation-based testing. For SPLs, mutation testing can be applied to the feature model, the mapping from features to source code (i.e., the presence conditions), and to the source code itself. [6] In the following, we will discuss related work starting with feature model mutations.

To mutate a feature model either the feature diagram can be mutated [16, 147, 65, 72, 131, 64] or the propositional formula of the feature diagram can be mutated [93, 95]. The resulting mutants of feature models have been subject to research and used to evaluate sampling effectiveness [64, 93, 95, 131, 72, 147] and generate samples based on the mutants [16, 147, 95, 131, 72]. In 2014 Lackner and Schmidt proposed a method to use feature model mutants to measure the effectiveness of model-based product-line testing techniques. [114]

In this thesis, feature-model mutations are currently out-of-scope. However, they might be incorporated easily. We embed the feature-model constraints as compound propositional formula (see Sect. 2.5). Therefore, we can use our mapping-mutation operators on the formula by applying mutation testing only to the corresponding function.

There also exists prior work on feature-mapping mutations. Al-Hajjaji et al. proposed a set of operators defined specifically to mutate `#ifdef` directives(e.g., `#ifdef` is changed to `#ifndef`). [6] Lackner and Schmidt defined mutation operators to mutate feature mappings in annotated UML state machines. [114] Although the mutation operators of Al-Hajjaji et al. and Lackner and Schmidt are defined differently from the mutation operators used in this thesis, the resulting mutations are similar to the resulting mutations used in our evaluation (e.g., negating a presence condition).

However, those works do not measure the effectiveness in terms of mutations detection of samples.

The standard use of mutation testing is by applying mutation operators for source-code transformation. For an overview of standard mutation-operators and their use, we refer to the work of Jia and Harman [103, 101], as well as the work of Offutt [137].

Papadakis et al. [141] and Bures and Ahmed [41] use standard mutation testing for single products to measure the effectiveness of combinatorial interaction testing (i.e., sampling the input parameters) for unit testing. In 2016 Al-Hajjaji et al. [6] proposed the use of standard-mutation operators for SPLs by applying the mutation operators to variable code blocks (e.g., `#ifdef` blocks). Dovrey et al. proposed the application of mutation testing to model-based SPLs and built mutant families to optimize efficiency during mutation testing.

In contrast to those works, our framework applies the mutation operators to SPLs implemented in C using runtime variability. Additionally, we combine both feature-mapping mutation operators and source-code mutation operators. Finally, our approach measure the effectiveness of samples by their potential to detect mutants by utilizing family-based test-case generation (i.e., independent of prior existing, potentially incomplete, test-suites for the samples).

Lastly, different works exist to improve SPL analysis. One possibility is to generate samples targeted to detect a given set of mutants [60, 9]. Other works are concerned with similar and equivalent mutants and their detection to enhance mutation testing in general [147, 43]. Lastly, variability analysis can be used to reduce the number of mutants that have to be analyzed during mutation testing [9].

In this work, potential improvements for mutation testing (e.g., detecting and removing equivalent mutants, reducing mutants using variability analyses, etc.) are out of scope. However, they might be incorporated in future work.

In this paper, we omit the potential improvements of mutant reduction and early detection of equivalent or similar mutants and leave the extensions open for future work.

5.8 Summary and Future Work

Conclusion. In this chapter, we presented a framework for measuring the effectiveness of SPL testing strategies. To this end, we utilized mutation testing techniques tailored for SPL implementations to simulate faults. Additionally, we showed existing and novel techniques for test-case generation used to generate test cases guaranteed to detect the mutants. These test cases were used to mea-

sure the effectiveness of samples based on the possibility of a sample to detect a mutant (i.e., if the sample contains a configuration on which the mutant is detectable). The results of our experimental evaluation gained from applying our framework to a collection of real-world and synthetic subject systems confirm the well-known assumption that pairwise sampling constitutes the best trade-off in terms of efficiency and effectiveness for sample-based SPL testing.

Future Work. For future work, we would like to extend our evaluation by measuring the effectiveness of product prioritization techniques [57, 18, 8, 158] and measure the effectiveness of different SPL testing techniques (e.g., family-based PL model checking [172]). Additionally, we would like to use our approach to generate samples achieving a predefined degree of mutation detection (i.e., effectiveness) for a given set of SPL mutants. Next, we plan to improve the mutation detection by a-priori removing equivalent mutants and reducing mutants using variability analyses. Additionally, we would like to tailor the mutant generation so that the distribution of mutants reflect the distribution of known SPL bugs (e.g., the distribution of feature-mapping bugs compared to the number of code-based bugs). Furthermore, we intend to incorporate feature model mutations into our framework to measure sampling effectiveness on feature model faults [95, 114, 147, 16]. Also, we plan to consider using larger subject systems (i.e., larger functions or system testing instead of unit testing) to further evaluate our approach. Lastly, we would also like to extend our approach to combine regression testing techniques from Chap. 4 with SPLs to increase effectiveness during regression testing of SPLs.

6

CONCLUSION AND FUTURE WORK

The goal of this thesis was the optimization of testing and test-case generation for software products, version histories, and software-product lines. To this end, we developed strategies for partitioning of test goals to optimize test-case generation. Furthermore, we build a framework incorporating new and existing techniques to provide a configurable test-case generation framework to optimize effectiveness and efficiency for regression testing. Lastly, we developed a novel technique for test-case generation for software-product lines and built a framework to measure the effectiveness of sample-based SPL testing.

In Chapter 1, we gave an overview of current challenges, and the research question tackled throughout this thesis. In the remainder of the conclusion, we summarize this thesis based on the research questions and summarize the future work and open challenges of the Chapters 3, 4 and 5.

6.0.1 *Summary*

Research Question 1 had the goal to optimize test-case generation using software model checking. Prior works have shown software model checking to be a promising technique for test-case generation. Additionally, there exist work to increase the $\mathit{efficiency}_{CPU}$ of test-case generation with software model checking. Apel et al. [15] were concerned with enhancing software model checking to support multi-property checking (i.e., to regard all test goals in a single reachability analysis). Jakobs was concerned with the increase of $\mathit{efficiency}_{CPU}$ by using hybrid model checking (i.e., to combine multiple analysis techniques in sequential order) also regarding all test goals at the same time. However, for software-model checking, there exist different techniques to abstract from unnecessary information during reachability analysis. For some test goals, the needed information differs. Therefore, running a single reachability analysis for all test goals might hinder abstraction and increase the CPU time.

In Chapter 3, we tackled this issue and improved the state of the art in two ways. First, we extended the CPAchecker framework by incorporating partitioning (i.e., grouping of test goals) into the test-case generation. Next, we developed multiple partitioning strategies based on local and global information about the test goals. We also evaluated the partitioning strategies for both the predicate and explicit value analysis. Additionally, we combined partitioning with hybrid model checking by combining the predicate and explicit value analysis with their respective best partitioning strategies. Lastly, we evaluated different hybrid model checking

strategies combined with the optimal partitioning strategies and without partitioning as baseline.

Research Question 2 had the goal to improve test-case generation for regression testing. The aim of regression testing is to detect bugs introduced by changes to the program. The only test cases that are able to detect new bugs are test cases that reveal changed behavior compared to the previous program version. Therefore, not all test cases are useful and some can be omitted during regression testing. To this end, different techniques have been proposed to make testing more focused on changes to the program, for example, by reducing the number of test cases in a test suite or by executing only certain test cases. [181] For test-case generation for regression testing, prior work also exists. For example, to generate test cases that reach at least one part of the changed code (called modification-traversing test cases) [70]. A more complex test-case selection criterion is modification-revealing test cases (also called differential test cases) and can also be used for test-case generation [70]. Those test cases not only reach the changes but also lead to different output behavior compared to the previous program version. However, there is still potential for optimization of regression testing, for example, by generating multiple test cases per test goal and, therefore, potentially increasing effectiveness.

In Chapter 4, we showed novel techniques to optimize test-case generation for regression testing. To this end, we developed a novel technique to generate multiple test cases per test goal that are guaranteed to traverse different program paths when executed. Additionally, we developed a framework incorporating this technique to allow for a configurable number of test cases generated per test goal. Additionally, we also take a configurable number of previous program versions into account and incorporated test-suite reduction techniques. Lastly, we evaluated our framework and studied the interaction between the different parameters on the effectiveness, efficiency, and the trade-off of both for regression testing.

Research Question 3 had the goal to optimize testing for software-product lines. As software-product lines are enriched with variability, standard software testing becomes insufficient. Due to the interaction between features, some bugs might only occur in specific products. Therefore, to fully test the SPL, the whole family of products needs to be tested. However, the number of products usually grows exponentially with the number of features, testing each product becomes infeasible. The most prominent technique to tackle this issue is sample-based testing. The aim of sample-based testing is to choose a preferably small, representative set of products that are tested. The challenge for testing of SPLs is, therefore, to choose a representative set of products (e.g., by τ-wise sampling) to optimize the trade-off between $\mathit{efficiency}_{\mathit{samplesize}}$ (i.e., the number of products that need to be tested) and effectiveness (the number of bugs found).

To this end, we developed a framework in Chapter 5 that utilizes mutation-based testing tailored for software-product lines and family-based test-case generation to evaluate samples based on their effectiveness. Next, we evaluated different sampling strategies based on their effectiveness (i.e., mutant-detection potential) and $\mathit{efficiency}_{\mathit{samplesize}}$ (i.e., sample size).

6.0.2 *Future Work*

For future work, there exist several open challenges and ideas to improve our frameworks and techniques.

For future work of **test-case generation for software products**, we would like to develop further partitioning strategies (e.g., based on data-flow analysis). Next, we would like to incorporate additional analysis techniques for hybrid test-case generation and also utilize different testing techniques (e.g., fuzzing) to further increase $efficiency_{CPU}$. Furthermore, we plan to evaluate on-the-fly partitioning (i.e., to re-partition after each reachability analysis) and use machine learning approaches to select the best partitioning strategy for the current program under test. The last and most promising future work for test-case generation for software products is to use partitioning strategies to parallelize test-case generation as the reachability analyses for the partitions can run on different cores or even distributed machines to increase the wall time of test-case generation.

In terms of **regression testing**, we would like to incorporate new techniques to our test-case generation approach to further improve $efficiency_{CPU}$ (e.g., fuzzing) and handle currently unsupported data types (e.g., arrays as return value) to support additional subject systems. Additionally, we would like to support the re-use of existing test-cases to further increase $efficiency_{CPU}$ and $efficiency_{size}$. Furthermore, we plan to evaluate the impact of different kinds of bugs (e.g., control-flow bugs and data-flow bugs) on the parameters of our framework. Next, we are interested in the impact of different coverage criteria for test-case generation on the effectiveness and efficiency of our framework. For example, test-cases reaching the modification and propagating the change, however, not necessarily revealing it. Finally, we plan to use our framework for different testing scenarios (e.g., integration testing or system testing) and to support different programming languages for test-case generation to extend the set of possible subject systems.

For future work of **Testing of Software-Product Lines**, we would like to measure the effectiveness of prioritization techniques [57, 18, 8, 158] and different SPL testing techniques [172]. Furthermore, we plan to use our approach to generate samples based on a predefined degree of mutation detection for a given set of SPL mutants. We also intend to improve our approach by removing equivalent mutants and, therefore, reducing the mutants under consideration. Additionally, we would like to study whether the distribution of the mutants reflects the distribution of known SPL bugs and, if not, adjust our mutant generation. We also plan to incorporate feature model mutations into our framework and measure the effectiveness of samples based on feature model faults. [95, 114, 147, 16] Next, we would like to consider larger functions and entire systems as subject systems to further evaluate our approach. Lastly, we would also like to extend our approach to combine regression testing techniques from Chap. 4 with SPLs to increase effectiveness during SPL regression testing.

Lastly, we would like to enhance our methodologies to handle and evaluate non-sequential code (i.e., parallelized or distributed software). Additionally, we would like to support more components of embedded software, such as interrupt handling, to support and evaluate a larger range of Industrie 4.0 software.

BIBLIOGRAPHY

[1] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. Variability bugs in highly configurable systems: A qualitative analysis. *ACM Trans. Softw. Eng. Methodol.*, 26:10:1–10:34, 2018. doi: https://doi.org/10.1145/3149119. (Cited on page 136.)

[2] Allen T. Acree, Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Mutation Analysis. Technical report, Georgia Inst of Tech Atlanta School of Information and Computer Science, 1979. (Cited on page 16.)

[3] Hiralal Agrawal, Richard A. DeMillo, Bob Hathaway, William Hsu, Wynne Hsu, E. W. Krauser, R. J. Martin, Aditya P. Mathur, and Eugene Spafford. Design of mutant operators for the c programming language – technical report. Technical report, 1989. (Cited on page 123.)

[4] Bestoun S. Ahmed, Kamal Z. Zamli, Wasif Afzal, and Miroslav Bures. Constrained interaction testing: A systematic literature study. *IEEE Access*, 5:25706–25730, 2017. doi: https://doi.org/10.1109/ACCESS.2017.2771562. (Cited on page 135.)

[5] Alfred Aho. *Compilers : principles, techniques, & tools*. Addison-Wesley, 2007. ISBN 0321491696. (Cited on page 81 and 83.)

[6] Mustafa Al-Hajjaji, Fabian Benduhn, Thomas Thüm, Thomas Leich, and Gunter Saake. Mutation Operators for Preprocessor-Based Variability. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, pages 81–88. ACM, 2016. doi: https://doi.org/10.1145/2866614.2866626. (Cited on page 136 and 137.)

[7] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 144–155. ACM, 2016. doi: https://doi.org/10.1145/2993236.2993253. (Cited on page 135.)

[8] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. Effective Product-Line Testing using Similarity-Based Product Prioritization. *Software & Systems Modeling*, 18:499–521, 2016. doi: https://doi.org/10.1007/s10270-016-0569-2. (Cited on page 138 and 141.)

[9] Mustafa Al-Hajjaji, Jacob Krüger, Fabian Benduhn, Thomas Leich, and Gunter Saake. Efficient Mutation Testing in Configurable Systems. In *2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software*

Design, pages 2–8. IEEE, 2017. doi: https://doi.org/10.1109/VACE.2017.3. (Cited on page 137.)

[10] Vitor Alcácer and Virgilio Cruz-Machado. Scanning the industry 4.0: A literature review on technologies for manufacturing systems. *Engineering Science and Technology, an International Journal*, 22:899–919, 2019. doi: https://doi.org/10.1016/j.jestch.2019.01.006. (Cited on page 7.)

[11] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2016. ISBN 978-1-107-17201-2. (Cited on page 11, 12, 16, 76, 123, and 124.)

[12] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? [software testing]. In *Proceedings. 27th International Conference on Software Engineering*, pages 402–411. IEEE, 2005. doi: https://doi.org/10.1109/ICSE.2005.1553583. (Cited on page 15, 92, and 103.)

[13] Pavel Andrianov, Alexey Khoroshilov, and Vadim Mutilin. An approach to lightweight static data race detection. pages 1–7. Institute for System Programming of the Russian Academy of Sciences, 2014. doi: https://doi.org/10.15514/syrcose-2014-8-4. (Cited on page 22.)

[14] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer Science and Business Media LLC, 2013. ISBN 978-3-642-37521-7. (Cited on page 1, 31, 33, and 34.)

[15] Sven Apel, Dirk Beyer, Vitaly Mordan, Vadim Mutilin, and Andreas Stahlbauer. On-the-fly decomposition of specifications in software model checking. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 349–361. ACM, 2016. doi: https://doi.org/10.1145/2950290.2950349. (Cited on page 2, 40, and 139.)

[16] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. Generating tests for detecting faults in feature models. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation*, pages 1–10. IEEE, 2015. doi: https://doi.org/10.1109/ICST.2015.7102591. (Cited on page 136, 138, and 141.)

[17] John Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. Regression Verification Using Impact Summaries. In *SPIN '13*, volume 7976, pages 99–116. Springer Berlin Heidelberg, 2013. doi: https://doi.org/10.1007/978-3-642-39176-7_7. (Cited on page 106.)

[18] Hauke Baller, Sascha Lity, Malte Lochau, and Ina Schaefer. Multi-objective test suite optimization for incremental product family testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 303–312. IEEE, 2014. doi: https://doi.org/10.1109/ICST.2014.43. (Cited on page 135, 138, and 141.)

[19] Animesh Basak Chowdhury, Raveendra Medicherla, and Rentala Venkatesh. *VeriFuzz: Program Aware Fuzzing: (Competition Contribution)*, pages 244–249. Springer International Publishing, 2019. ISBN 978-1-4939-9100-6. doi: https://doi.org/10.1007/978-3-030-17502-3_22. (Cited on page 66.)

[20] Thomas Bauernhansl, Michael Hompel, and Birgit Vogel-Heuser. *Industrie 4.0 in Produktion, Automatisierung und Logistik: Anwendung - Technologien - Migration*. Springer Vieweg, 2014. ISBN 978-3-658-04681-1. (Cited on page 7.)

[21] Dirk Beyer. Configurable software model checking". `https://www.sosy-lab.org/research/prs/Current_CPAchecker.pdf`. Accessed: 20.09.2021. (Cited on page 19, 21, 52, 87, and 126.)

[22] Dirk Beyer. Benchmarks: Benchmark set of 8th intl. competition on software verification. https://doi.org/10.5281/zenodo.2598728, 2019. (Cited on page 102.)

[23] Dirk Beyer. International competition on software testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 11429, pages 167–175. Springer International Publishing, 2019. doi: https://doi.org/10.1007/978-3-030-17502-3_11. (Cited on page 52.)

[24] Dirk Beyer and Marie-Christine Jakobs. Coveritest: Cooperative verifier-based testing. In *Proceedings of the 22nd International Conference on Fundamental Approaches to Software Engineering*, volume 11424, pages 389–408. Springer International Publishing, 2019. doi: https://doi.org/10.1007/978-3-030-16722-6_23. (Cited on page 3, 5, 51, and 67.)

[25] Dirk Beyer and Marie-Christine Jakobs. Cooperative verifier-based testing with CoVeriTest. *International Journal on Software Tools for Technology Transfer*, 21:313–333, 2021. doi: https://doi.org/10.1007/s10009-020-00587-8. (Cited on page 51 and 67.)

[26] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification*, volume 6806, pages 184–190. Springer Berlin Heidelberg, 2011. doi: https://doi.org/10.1007/978-3-642-22110-1_16. (Cited on page 21, 22, and 26.)

[27] Dirk Beyer and Thomas Lemberger. Symbolic execution with cegar. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, volume 9952, pages 195–211. Springer International Publishing, 2016. doi: https://doi.org/10.1007/978-3-319-47166-2_14. (Cited on page 66.)

[28] Dirk Beyer and Thomas Lemberger. Conditional testing - off-the-shelf combination of test-case generators. In *Proceedings of the 17th International Symposium on Automated Technology for Verification and Analysis*, volume 11781, pages 189–208. Springer International Publishing, 2019. doi: https://doi.org/10.1007/978-3-030-31784-3_11. (Cited on page 51.)

[29] Dirk Beyer and Thomas Lemberger. Testcov: Robust test-suite execution and coverage measurement. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, page 1074–1077. IEEE, 2019. doi: https://doi.org/10.1109/ASE.2019.00105. (Cited on page 71, 74, 86, and 103.)

[30] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*, pages 326–335. IEEE, 2004. doi: https://doi.org/10.1109/ICSE.2004.1317455. (Cited on page 2, 17, and 24.)

[31] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9:505–525, 2007. doi: https://doi.org/10.1007/s10009-007-0044-z. (Cited on page 17.)

[32] Dirk Beyer, Thomas A. Henzinger, and Gregory Theoduloz. Program analysis with dynamic precision adjustment. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 29–38. IEEE, 2008. doi: https://doi.org/10.1109/ASE.2008.13. (Cited on page 23.)

[33] Dirk Beyer, Andreas Holzer, Michael Tautschnig, and Helmut Veith. Information reuse for multi-goal reachability analyses. In *ESOP '13*, volume 7792, pages 472–491. Springer Berlin Heidelberg, 2013. doi: https://doi.org/10.1007/978-3-642-37036-6_26. (Cited on page 17 and 26.)

[34] Dirk Beyer, Stefan Löwe, Evgeny Novikov, Andreas Stahlbauer, and Philipp Wendler. Precision Reuse for Efficient Regression Verification. In *ESEC/FSE '13*, pages 389–399. ACM, 2013. doi: https://doi.org/10.1145/2491411.2491429. (Cited on page 106.)

[35] Dirk Beyer, Sumit Gulwani, and David A. Schmidt. Combining model checking and data-flow analysis. In *Handbook of Model Checking*, volume 1, pages 493–540. Springer International Publishing, 2018. doi: https://doi.org/10.1007/978-3-319-10575-8_16. (Cited on page 22 and 23.)

[36] Dirk Beyer, Marie-Christine Jakobs, Thomas Lemberger, and Heike Wehrheim. Combining verifiers in conditional model checking via reducers. In *Proceedings of the Conference on Software Engineering and Software Management*, pages 151–152. GI, 2019. doi: https://doi.org/10.18420/se2019-46. (Cited on page 51.)

[37] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. Partition-based Regression Verification. In *ICSE '13*, pages 302–311. IEEE, 2013. doi: https://doi.org/10.1109/ICSE.2013.6606576. (Cited on page 106.)

[38] Birgit Boss, Sebastian Bader, Andreas Orzelski, and Michael Hoffmeister. Verwaltungsschale. In *Springer Reference Technik*, pages 1–27. Springer Berlin Heidelberg, 2019. doi: https://doi.org/10.1007/978-3-662-45537-1_139-1. (Cited on page 9 and 10.)

[39] Johannes Bürdek. *Rekonfigurierbare Software-Systeme: Spezifikation und Testfallgenerierung*. PhD thesis, Darmstadt University of Technology, Germany, 2018. (Cited on page 115.)

[40] Johannes Bürdek, Malte Lochau, Stefan Bauregger, Andreas Holzer, Alexander von Rhein, Sven Apel, and Dirk Beyer. Facilitating reuse in multi-goal test-suite generation for software product lines. In *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering*, volume 9033, pages 84–99. Springer Berlin Heidelberg, 2015. doi: https://doi.org/10.1007/978-3-662-46675-9_6. (Cited on page 5, 24, 37, and 113.)

[41] Miroslav Bures and Bestoun S. Ahmed. On the Effectiveness of Combinatorial Interaction Testing: A Case Study. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion*, pages 69–76. IEEE, 2017. doi: https://doi.org/10.1109/QRS-C.2017.20. (Cited on page 137.)

[42] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 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*, volume 8, page 209–224. USENIX Association, 2008. (Cited on page 66.)

[43] Luiz Carvalho, Marcio Augusto Guimarães, Márcio Ribeiro, Leonardo Fernandes, Mustafa Al-Hajjaji, Rohit Gheyi, and Thomas Thüm. Equivalent Mutants in Configurable Systems: An Empirical Study. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, pages 11–18. ACM, 2018. doi: https://doi.org/10.1145/3168365.3168379. (Cited on page 137.)

[44] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Regression Verification for Multi-threaded Programs. In *VMCAI '12*, volume 7148, pages 119–135. Springer Berlin Heidelberg, 2012. doi: https://doi.org/10.1007/978-3-642-27940-9_9. (Cited on page 106.)

[45] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, and Bing Xie. How do assertions impact coverage-based test-suite reduction? In *2017 IEEE International Conference on Software Testing, Verification and Validation*, pages 418–423. IEEE, 2017. doi: https://doi.org/10.1109/icst.2017.45. (Cited on page 104.)

[46] Tsong Yueh Chen and Man Fai Lau. Dividing strategies for the optimization of a test suite. *Inf. Process. Lett.*, 60:135–141, 1996. doi: https://doi.org/10.1016/S0020-0190(96)00135-4. (Cited on page 103.)

[47] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. SAVIOR: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy*, pages 1580–1596. IEEE, 2020. doi: https://doi.org/10.1109/sp40000.2020.00002. (Cited on page 66.)

[48] Philippe Chevalley. Applying mutation analysis for object-oriented programs using a reflective approach. In *Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*, pages 267–270. IEEE, 2001. doi: https://doi.org/10.1109/APSEC.2001.991487. (Cited on page 16.)

[49] Ankur Choudhary, Arun Prakash Agrawal, and Arvinder Kaur. An effective approach for regression test case selection using pareto based multi-objective harmony search. In *Proceedings of the 11th International Workshop on Search-Based Software Testing*, pages 13–20. ACM, 2018. doi: https://doi.org/10.1145/3194718.3194722. (Cited on page 104.)

[50] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50:752–794, 2003. doi: https://doi.org/10.1145/876638.876643. (Cited on page 25.)

[51] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. ISBN 978-0201703320. (Cited on page 31 and 109.)

[52] Anastasia Cmyrev and Ralf Reissing. Efficient and Effective Testing of Automotive Software Product Lines. *IJAST*, 7:53–57, 2014. doi: https://doi.org/10.14416/j.ijast.2014.05.001. (Cited on page 135.)

[53] Emilio Cruciani, Breno Miranda, Roberto Verdecchia, and Antonia Bertolino. Scalable approaches for test suite reduction. In *2019 IEEE/ACM 41st International Conference on Software Engineering*, pages 419–429. IEEE, 2019. doi: https://doi.org/10.1109/ICSE.2019.00055. (Cited on page 71 and 103.)

[54] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. Cool features and tough decisions: A comparison of variability modeling approaches. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, pages 173–182. Association for Computing Machinery, 2012. doi: https://doi.org/10.1145/2110147.2110167. (Cited on page 34.)

[55] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, John D. McGregor, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. A Systematic Mapping Study of Software Product Lines Testing. *Inf. Softw. Technol.*, 53:407–423, 2011. doi: https://doi.org/10.1016/j.infsof.2010.12.003. (Cited on page 110, 112, and 135.)

[56] Richard DeMillo, Richard Lipton, and Frederick Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11:34–41, 1978. doi: https://doi.org/10.1109/C-M.1978.218136. (Cited on page 15.)

[57] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Pierre-Yves Schobbens, Axel Legay, and Patrick Heymans. Towards Statistical Prioritization for Software Product Lines Testing. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, pages 10:1–10:7. ACM,

2013. doi: https://doi.org/10.1145/2556624.2556635. (Cited on page 138 and 141.)

[58] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Covering SPL Behaviour with Sampled Configurations: An Initial Assessment. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*, pages 59:59–59:66. ACM, 2015. doi: https://doi.org/10.1145/2701319.2701325. (Cited on page 136.)

[59] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Search-Based Similarity-Driven Behavioural SPL Testing. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, pages 89–96. ACM, 2016. doi: https://doi.org/10.1145/2866614.2866627. (Cited on page 136.)

[60] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Featured Model-based Mutation Analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 655–666. ACM, 2016. doi: https://doi.org/10.1145/2884781.2884821. (Cited on page 137.)

[61] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques. *Empirical Software Engineering*, 10:405–435, 2005. doi: https://doi.org/10.1007/s10664-005-3861-2. (Cited on page 103.)

[62] Ivan do Carmo Machado, John D. McGregor, and Eduardo Santana de Almeida. Strategies for Testing Products in Software Product Lines. *SIGSOFT Softw. Eng. Notes*, 37:1–8, 2012. doi: https://doi.org/10.1145/2382756.2382783. (Cited on page 4, 36, and 109.)

[63] Ivan do Carmo Machado, John D. McGregor, Yguarata Cerqueira Cavalcanti, and Eduardo Santana de Almeida. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *Information and Software Technology*, 56:1183 – 1199, 2014. doi: https://doi.org/10.1016/j.infsof.2014.04.002. (Cited on page 4, 110, 112, and 135.)

[64] Thiago do Nascimento Ferreira, Josiel Neumann Kuk, Aurora Pozo, and Silvia Regina Vergilio. Product selection based on upper confidence bound moea/d-dra for testing software product lines. In *2016 IEEE Congress on Evolutionary Computation*, pages 4135–4142. IEEE, 2016. doi: https://doi.org/10.1109/CEC.2016.7744315. (Cited on page 135 and 136.)

[65] Thiago do Nascimento Ferreira, Jackson A. Prado Lima, Andrei Strickler, Josiel Neumann Kuk, Silvia Regina Vergilio, and Aurora Trinidad Ramirez Pozo. Hyper-heuristic based product selection for software product line testing. *IEEE Computational Intelligence Magazine*, 12:34–45, 2017. doi: https://doi.org/10.1109/MCI.2017.2670461. (Cited on page 135 and 136.)

[66] Sebastian Elbaum, David Gable, and Gregg Rothermel. Understanding and measuring the sources of variation in the prioritization of regression test suites. In *Proceedings Seventh International Software Metrics Symposium*, pages 169–179. IEEE, 2001. doi: https://doi.org/10.1109/METRIC.2001.915525. (Cited on page 105.)

[67] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 329–338. IEEE, 2001. doi: https://doi.org/10.1109/ICSE.2001.919106. (Cited on page 105.)

[68] Emelie Engström and Per Runeson. Software Product Line Testing - A Systematic Mapping Study. *Information and Software Technology*, 53:2 – 13, 2011. doi: https://doi.org/10.1016/j.infsof.2010.05.011. (Cited on page 110, 112, and 135.)

[69] Faezeh Ensan, Ebrahim Bagheri, and Dragan Gašević. Evolutionary Search-Based Test Generation for Software Product Line Feature Models. In *Advanced Information Systems Engineering*, volume 7328, pages 613–628. Springer Berlin Heidelberg, 2012. doi: https://doi.org/10.1007/978-3-642-31095-9_40. (Cited on page 135.)

[70] Robert B. Evans and Alberto Savoia. Differential testing: A new approach to change detection. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, pages 549–552. ACM, 2007. doi: https://doi.org/10.1145/1295014.1295038. (Cited on page 3, 105, and 140.)

[71] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating Regression Verification. In *ASE '14*, pages 349–360. ACM, 2014. doi: https://doi.org/10.1145/2642937.2642987. (Cited on page 106.)

[72] Helson L. Jakubovski Filho, Jackson A. Prado Lima, and Silvia R. Vergilio. Automatic generation of search-based algorithms applied to the feature testing of software product lines. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, pages 114–123. ACM, 2017. doi: https://doi.org/10.1145/3131151.3131152. (Cited on page 135 and 136.)

[73] Stefan Fischer, Roberto E. Lopez-Herrejon, Rudolf Ramler, and Alexander Egyed. A Preliminary Empirical Assessment of Similarity for Combinatorial Iteraction Testing of Software Product Lines. In *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing*, pages 15–18. ACM, 2016. doi: https://doi.org/10.1145/2897010.2897011. (Cited on page 135 and 136.)

[74] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software*

Testing and Analysis, pages 147–158. ACM, 2010. doi: https://doi.org/10.1145/1831708.1831728. (Cited on page 106.)

[75] Mikhail R. Gadelha, Rafael Menezes, Felipe R. Monteiro, Lucas C. Cordeiro, and Denis Nicole. ESBMC: Scalable and precise test generation based on the floating-point theory. In *Proceedings of the 23rd International Conference on Fundamental Approaches to Software Engineering*, pages 525–529. Springer International Publishing, 2020. doi: https://doi.org/10.1007/978-3-030-45234-6_27. (Cited on page 66.)

[76] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. Evaluating Improvements to a Meta-Heuristic Search for Constrained Interaction Testing. *Empirical Software Engineering*, 16:61–102, 2011. doi: https://doi.org/10.1007/s10664-010-9135-7. (Cited on page 135.)

[77] ACX GmbH. DO-178C Introduction. `https://acx-gmbh.de/de/aviation.html?file=files/acx_layout/downloads/DO178C.pdf`. Accessed: 29.11.2021. (Cited on page 2.)

[78] Sangharatna Godboley, G.S. Prashanth, Durga Mohapatra, and Banshidhar Majhi. Increase in modified condition/decision coverage using program code transformer. In *2013 3rd IEEE International Advance Computing Conference*, pages 1400–1407. IEEE, 2013. doi: https://doi.org/10.1109/IAdCC.2013.6514432. (Cited on page 40.)

[79] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 43, page 206–215. ACM, 2008. doi: https://doi.org/10.1145/1375581.1375607. (Cited on page 65.)

[80] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium*, 2008. (Cited on page 65.)

[81] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *Proceedings Conference on Software Maintenance 1992*, pages 299–308. IEEE, 1992. doi: https://doi.org/10.1109/ICSM.1992.242531. (Cited on page 104.)

[82] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. Test Them All, Is It Worth It? Assessing Configuration Sampling on the JHipster Web Development Stack. *Empirical Software Engineering*, 24:674–717, 2017. doi: https://doi.org/10.1007/s10664-018-9635-4. (Cited on page 136.)

[83] Grégoire Hamon, Leonardo de Moura, and John Rushby. Generating efficient test sets with a model checker. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004.*, pages 261–270. IEEE, 2004. doi: https://doi.org/10.1109/SEFM.2004.1347530. (Cited on page 2.)

[84] Mark Harman, Yue Jia, and William B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 212–222. ACM, 2011. doi: https://doi.org/10.1145/2025113.2025144. (Cited on page 106.)

[85] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2 edition, 2016. doi: 10.1017/CBO9781316576892. (Cited on page 83.)

[86] Mary Jean Harrold and Mary Lou Soffa. Interprocedual data flow testing. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*, volume 14, pages 158–167. ACM, 1989. doi: https://doi.org/10.1145/75308.75327. (Cited on page 104.)

[87] Mary Jean Harrold and Mary Lou Souffa. An incremental approach to unit testing during maintenance. In *Proceedings. Conference on Software Maintenance, 1988.*, pages 362–367. IEEE, 1988. doi: https://doi.org/10.1109/ICSM.1988.10188. (Cited on page 104.)

[88] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2:270–285, 1993. doi: https://doi.org/10.1145/152388.152391. (Cited on page 28 and 103.)

[89] Jean Hartmann and David J. Robson. Retest-development of a selective revalidation prototype environment for use in software maintenance. In *Twenty-Third Annual Hawaii International Conference on System Sciences*, volume 2, pages 92–101 vol.2. IEEE, 1990. doi: https://doi.org/10.1109/HICSS.1990.205179. (Cited on page 104.)

[90] Jean Hartmann and David J. Robson. Techniques for selective revalidation. *IEEE Software*, 7:31–36, 1990. doi: https://doi.org/10.1109/52.43047. (Cited on page 104.)

[91] Jean Zoren Werner Hartmann and David J. Robson. Revalidation during the software maintenance phase. In *Proceedings. Conference on Software Maintenance - 1989*, pages 70–80. IEEE, 1989. doi: https://doi.org/10.1109/ICSM.1989.65195. (Cited on page 104.)

[92] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, and Alexander Egyed. Using feature model knowledge to speed up the generation of covering arrays. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, pages 16:1–16:6. ACM, 2013. doi: https://doi.org/10.1145/2430502.2430524. (Cited on page 135.)

[93] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Assessing software product line testing via model-based mutation: An application to similarity testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*,

pages 188–197. IEEE, 2013. doi: https://doi.org/10.1109/ICSTW.2013.30. (Cited on page 136.)

[94] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Multi-objective test generation for software product lines. In *Proceedings of the 17th International Software Product Line Conference*, pages 62–71. ACM, 2013. doi: https://doi.org/10.1145/2491627.2491635. (Cited on page 135.)

[95] Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutation-based generation of software product line test configurations. In *Search-Based Software Engineering*, volume 8636, pages 92–106. Springer International Publishing Switzerland, 2014. doi: https://doi.org/10.1007/978-3-319-09940-8_7. (Cited on page 136, 138, and 141.)

[96] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering*, 40: 650–670, 2014. doi: https://doi.org/10.1109/TSE.2014.2327020. (Cited on page 135.)

[97] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Marco A. A. Sanvido. *Extreme Model Checking*, volume 2772 of *LNCS*, pages 332–358. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-39910-0. doi: https://doi.org/10.1007/978-3-540-39910-0_16. (Cited on page 106.)

[98] Andreas Holzer, Michael Tautschnig, Christian Schallhart, and Helmut Veith. An introduction to test specification in fql. In *Hardware and Software: Verification and Testing*, volume 6504, pages 9–22. Springer Berlin Heidelberg, 2011. doi: https://doi.org/10.1007/978-3-642-19583-9_5. (Cited on page 115.)

[99] Joseph R. Horgan and Saul London. A data flow coverage testing tool for c. In *[1992] Proceedings of the Second Symposium on Assessment of Quality Software Development Tools*, pages 2–10. IEEE, 1992. doi: https://doi.org/10.1109/AQSDT.1992.205829. (Cited on page 103.)

[100] Marie-Christine Jakobs. CoVeriTest with dynamic partitioning of the iteration time limit (competition contribution). In *Proceedings of the 23rd International Conference on Fundamental Approaches to Software Engineering*, volume 12076, pages 540–544. Springer International Publishing, 2020. doi: https://doi.org/10.1007/978-3-030-45234-6_30. (Cited on page 41, 51, and 66.)

[101] Yue Jia and Mark Harman. *Mutation Testing Repository*. URL `http://crestweb.cs.ucl.ac.uk/resources/mutation_testing_repository/`. Accessed: 06.06.2018. (Cited on page 137.)

[102] Yue Jia and Mark Harman. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In *Testing:*

Academic Industrial Conference - Practice and Research Techniques, pages 94–98. IEEE, 2008. doi: https://doi.org/10.1109/TAIC-PART.2008.18. (Cited on page 123.)

[103] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37:649–678, 2011. doi: https://doi.org/10.1109/TSE.2010.62. (Cited on page 106, 122, 124, and 137.)

[104] Martin Johansen, Øystein Haugen, Franck Fleurey, Anne Eldegard, and Torbjørn Syversen. Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines. *Model Driven Engineering Languages and Systems*, 7590:269–284, 2012. doi: https://doi.org/10.1007/978-3-642-33666-9_18. (Cited on page 135.)

[105] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. Properties of realistic feature models make combinatorial testing of product lines feasible. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, pages 638–652. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-24485-8_47. (Cited on page 135.)

[106] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, volume 1, pages 46–55. ACM, 2012. doi: https://doi.org/10.1145/2362536.2362547. (Cited on page 135.)

[107] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990. URL `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231`. (Cited on page 34 and 135.)

[108] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward variability-aware testing. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, page 1–8. ACM, 2012. doi: https://doi.org/10.1145/2377816.2377817. (Cited on page 113.)

[109] Alireza Khalilian and Saeed Parsa. Bi-criteria test suite reduction by cluster analysis of execution profiles. In *Advances in Software Engineering Techniques*, volume 7054, pages 243–256. Springer Berlin Heidelberg, 2012. doi: https://doi.org/10.1007/978-3-642-28038-2_19. (Cited on page 71, 73, and 103.)

[110] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, pages 57–68. ACM, 2011. doi: https://doi.org/10.1145/1960275.1960284. (Cited on page 135.)

[111] Bogdan Korel and Ali M. Al-Yami. Automated regression test generation. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, volume 23, pages 143–152. ACM, 1998. doi: https://doi.org/10.1145/271771.271803. (Cited on page 105.)

[112] D. Richard Kuhn, Dolores R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30:418–421, 2004. doi: https://doi.org/10.1109/TSE.2004.24. (Cited on page 112.)

[113] Vladimir Kutscher, Sebastian Ruland, Patrick Müller, Nathan Wasser, Malte Lochau, Reiner Anderl, Andy Schürr, Mira Mezini, and Reiner Hähnle. Towards a circular economy of industrial software. *Procedia CIRP*, 90:37–42, 2020. doi: https://doi.org/10.1016/j.procir.2020.01.133. (Cited on page 8 and 9.)

[114] Hartmut Lackner and Martin Schmidt. Towards the assessment of software product line tests: A mutation system for variable systems. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*, volume 2, pages 62–69. ACM, 2014. doi: https://doi.org/10.1145/2647908.2655968. (Cited on page 136, 137, 138, and 141.)

[115] Jihyun Lee, Sungwon Kang, and Danhyung Lee. A Survey on Software Product Line Testing. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, volume 1, pages 31–40. ACM, 2012. doi: https://doi.org/10.1145/2362536.2362545. (Cited on page 110, 112, and 135.)

[116] Stefan-Helmut Leitner and W. Mahnke. Opc ua - service-oriented architecture for industrial applications. *Softwaretechnik-Trends*, 26:6, 2006. (Cited on page 9.)

[117] Thomas Lemberger. Plain random test generation with PRTest. *International Journal on Software Tools for Technology Transfer*, 2020. doi: https://doi.org/10.1007/s10009-020-00568-x. (Cited on page 66.)

[118] Hareton Leung and Lee White. Insights into regression testing (software testing). *Proceedings. Conference on Software Maintenance - 1989*, pages 60–69, 1989. doi: https://doi.org/10.1109/ICSM.1989.65194. (Cited on page 29.)

[119] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1, 2018. doi: https://doi.org/10.1186/s42400-018-0002-y. (Cited on page 65.)

[120] Romulo G. Lins, Bruno Guerreiro, Robert Schmitt, Jianing Sun, Marcio Corazzim, and Francis R. Silva. A novel methodology for retrofitting cnc machines based on the context of industry 4.0. In *2017 IEEE International Symposium on Systems Engineering*, pages 1–6. IEEE, 2017. doi: https://doi.org/10.1109/SysEng.2017.8088293. (Cited on page 7.)

[121] Theo Lins and Ricardo Augusto Rabelo Oliveira. Cyber-physical production systems retrofitting in context of industry 4.0. *Computers & Industrial Engineering*, 139:106193, 2020. doi: https://doi.org/10.1016/j.cie.2019.106193. (Cited on page 7.)

[122] Theo Lins, Ricardo Augusto Rabelo Oliveira, Luiz H. A. Correia, and Jorge Sa Silva. Industry 4.0 retrofitting. In *2018 VIII Brazilian Symposium on Computing Systems Engineering*, pages 8–15. IEEE, 05.11.2018 - 08.11.2018. doi: https://doi.org/10.1109/SBESC.2018.00011. (Cited on page 7.)

[123] Malte Lochau. *Model-Based Conformance Testing of Software Product Lines*. PhD thesis, University of Braunschweig - Institute of Technology, 2013. (Cited on page 35.)

[124] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental model-based testing of delta-oriented software product lines. In *Tests and Proofs - 6th International Conference, TAP 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings*, volume 7305, pages 67–82. Springer Berlin Heidelberg, 2012. doi: https://doi.org/10.1007/978-3-642-30473-6_7. (Cited on page 36.)

[125] Roberto E. Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Alexander Egyed, and Enrique Alba. Comparative Analysis of Classical Multi-Objective Evolutionary Algorithms and Seeding Strategies for Pairwise Testing of Software Product Lines. In *2014 IEEE Congress on Evolutionary Computation*, pages 387–396. IEEE, 2014. doi: https://doi.org/10.1109/CEC.2014.6900473. (Cited on page 135.)

[126] Roberto E. Lopez-Herrejon, Stefan Fischer, Rudolf Ramler, and Alexander Egyed. A First Systematic Mapping Study on Combinatorial Interaction Testing for Software Product Lines. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops*, pages 1–10. IEEE, 2015. doi: https://doi.org/10.1109/ICSTW.2015.7107435. (Cited on page 4, 112, and 135.)

[127] Roberto E. Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Alexander Egyed, and Enrique Alba. *Evolutionary Computation for Software Product Line Testing: An Overview and Open Challenges*, pages 59–87. Springer International Publishing, 2016. ISBN 978-3-319-25964-2. doi: https://doi.org/10.1007/978-3-319-25964-2_4. URL `https://doi.org/10.1007/978-3-319-25964-2_4`. (Cited on page 135.)

[128] Alexey Malishevsky, Gregg Rothermel, and Sebastian Elbaum. Modeling the cost-benefits tradeoffs for regression testing techniques. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 204–213. IEEE, 2002. doi: https://doi.org/10.1109/ICSM.2002.1167767. (Cited on page 105.)

[129] Dusica Marijan and Marius Liaaen. Practical selective regression testing with effective redundancy in interleaved tests. In *Proceedings of the 40th In-*

ternational Conference on Software Engineering: Software Engineering in Practice, page 153–162. ACM, 2018. doi: https://doi.org/10.1145/3183519.3183532. (Cited on page 104.)

[130] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. Practical Pairwise Testing for Software Product Lines. In *Proceedings of the 17th International Software Product Line Conference*, pages 227–235. ACM, 2013. doi: https://doi.org/10.1145/2491627.2491646. (Cited on page 135.)

[131] Rui A. Matnei Filho and Silvia R. Vergilio. A multi-objective test data generation approach for mutation testing of feature models. *Journal of Software Engineering Research and Development*, 4:4, 2016. doi: https://doi.org/10.1186/s40411-016-0030-9. (Cited on page 136.)

[132] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10:100–107, 1998. (Cited on page 30 and 105.)

[133] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering*, pages 643–654. ACM, 2016. doi: https://doi.org/10.1145/2884781.2884793. (Cited on page 136.)

[134] Elisa Negri, Luca Fumagalli, and Marco Macchi. A review of the roles of digital twin in cps-based production systems. *Procedia Manufacturing*, 11: 939–948, 2017. doi: https://doi.org/10.1016/j.promfg.2017.07.198. (Cited on page 10.)

[135] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1:5–20, 1992. doi: https://doi.org/10.1145/125489.125473. (Cited on page 15.)

[136] A. Jefferson Offutt, Jie Pan, and Jeffrey M. Voas. Procedures for reducing the size of coverage-based test sets. In *Twelfth International Conference on Testing Computer Software*, pages 111–123, 1995. (Cited on page 103.)

[137] Jeff Offutt. A Mutation Carol: Past, Present and Future. *Information and Software Technology*, 53:1098 – 1107, 2011. doi: https://doi.org/10.1016/j.infsof.2011.03.007. (Cited on page 137.)

[138] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Software Product Lines: Going Beyond*, volume 6287, pages 196–210. Springer Berlin Heidelberg, 2010. doi: https://doi.org/10.1007/978-3-642-15579-6_14. (Cited on page 135.)

[139] Sebastian Oster, Andreas Wübbeke, Gregor Engels, and Andy Schürr. A Survey of Model-Based Software Product Lines Testing. In *Model-based Testing for Embedded Systems*, pages 338 – 381. CRC Press, 2011. (Cited on page 110, 112, and 135.)

[140] Sebastian Oster, Marius Zink, Malte Lochau, and Mark Grechanik. Pairwise feature-interaction testing for spls: Potentials and limitations. In *Proceedings of the 15th International Software Product Line Conference*, volume 2, pages 6:1–6:8. ACM, 2011. doi: https://doi.org/10.1145/2019136.2019143. (Cited on page 135.)

[141] Mike Papadakis, Christopher Henard, and Yves Le Traon. Sampling Program Inputs with Mutation Analysis: Going Beyond Combinatorial Interaction Testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 1–10. IEEE, 2014. doi: https://doi.org/10.1109/ICST.2014.11. (Cited on page 137.)

[142] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 459–468. IEEE, 2010. doi: https://doi.org/10.1109/ICST.2010.43. (Cited on page 135.)

[143] Duy Loc Phan, Yunho Kim, and Moonzoo Kim. MUSIC: mutation analysis tool with high configurability and extensibility. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 40–46. IEEE, 2018. doi: https://doi.org/10.1109/ICSTW.2018.00026. (Cited on page 92.)

[144] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Science & Business Media, 2005. ISBN 978-3-540-28901-2. (Cited on page 1 and 33.)

[145] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *Proceedings 2017 Network and Distributed System Security Symposium*, pages 1–14. Internet Society, 2017. doi: https://doi.org/10.14722/ndss.2017.23404. (Cited on page 65.)

[146] GRBL Repository. Grbl v1.1 Interface. `https://github.com/gnea/grbl/wiki/Grbl-v1.1-Interface`, 2021. Accessed: 20.09.2021. (Cited on page 10.)

[147] Dennis Reuling, Johannes Bürdek, Serge Rotärmel, Malte Lochau, and Udo Kelter. Fault-based Product-line Testing: Effective Sample Generation Based on Feature-diagram Mutation. In *Proceedings of the 19th International Conference on Software Product Line*, pages 131–140. ACM, 2015. doi: https://doi.org/10.1145/2791060.2791074. (Cited on page 136, 137, 138, and 141.)

[148] John S Rinaldi. *OPC UA - Unified Architecture: The Everyman's Guide to the Most Important Information Technology in Industrial Automation*. CreateSpace Independent Publishing Platform, 2016. ISBN 978-1530505111. (Cited on page 9.)

[149] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, page 12–27. ACM, 1988. doi: https://doi.org/10.1145/73560.73562. (Cited on page 19.)

[150] Roland Rosen, Georg von Wichert, George Lo, and Kurt D. Bettenhausen. About the importance of autonomy and digital twins for the future of manufacturing. *IFAC-PapersOnLine*, 48:567–572, 2015. doi: https://doi.org/10.1016/j.ifacol.2015.06.141. (Cited on page 9.)

[151] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: an empirical study. In *Proceedings IEEE International Conference on Software Maintenance - 1999*, pages 179–188. IEEE, 1999. doi: https://doi.org/10.1109/ICSM.1999.792604. (Cited on page 105.)

[152] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27:929–948, 2001. doi: https://doi.org/10.1109/32.962562. (Cited on page 105.)

[153] Gregg Rothermel, Sebastian Elbaum, Alexey Malishevsky, Praveen Kallakuri, and Brian Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 130–140. ACM, 2002. doi: https://doi.org/10.1145/581356.581358. (Cited on page 105.)

[154] Sebastian Ruland and Malte Lochau. On the interaction between test-suite reduction and regression-test selection strategies. Technical report, 2022. URL `https://doi.org/10.48550/arXiv.2207.12733`. (Cited on page 6, 11, 27, 32, 70, 72, 73, 82, 92, 96, 97, 98, 99, and 111.)

[155] Sebastian Ruland, Lars Luthmann, Johannes Bürdek, Sascha Lity, Thomas Thüm, Malte Lochau, and Márcio Ribeiro. Measuring Effectiveness of Sample-Based Product-line Testing. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 119–133. ACM, 2018. doi: https://doi.org/10.1145/3278122.3278130. (Cited on page 6, 32, 35, 110, 111, 121, 122, 123, 129, 131, and 132.)

[156] Sebastian Ruland, Malte Lochau, Oliver Fehse, and Andy Schürr. CPA/Tiger-MGP: test-goal set partitioning for efficient multi-goal test-suite generation. *International Journal on Software Tools for Technology Transfer*, pages 1–4, 2020. doi: https://doi.org/10.1007/s10009-020-00574-z. (Cited on page 6, 39, and 43.)

[157] Sebastian Ruland, Malte Lochau, and Marie-Christine Jakobs. HybridTiger: Hybrid Model Checking and Domination-based Partitioning for Efficient Multi-Goal Test-Suite Generation (Competition Contribution). In *Fundamental Approaches to Software Engineering*, volume 12076, pages 520–524.

Springer International Publishing, 2020. doi: https://doi.org/10.1007/978-3-030-45234-6_26. (Cited on page 6 and 39.)

[158] Ana B. Sánchez, Sergio Segura, and Antonio Ruiz-Cortés. A Comparison of Test Case Prioritization Criteria for Software Product Lines. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 41–50. IEEE, 2014. doi: https://doi.org/10.1109/ICST.2014.15. (Cited on page 136, 138, and 141.)

[159] Ana B. Sánchez, Sergio Segura, José A. Parejo, and Antonio Ruiz-Cortés. Variability Testing in the Wild: The Drupal Case Study. *Software & Systems Modeling*, 16:173–194, 2017. doi: https://doi.org/10.1007/s10270-015-0459-z. (Cited on page 136.)

[160] Jan Schlechtendahl, Matthias Keinert, Felix Kretschmer, Armin Lechler, and Alexander Verl. Making existing production systems industry 4.0-ready. *Production Engineering*, 9:143–148, 2015. doi: https://doi.org/10.1007/s11740-014-0586-3. (Cited on page 7.)

[161] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017. doi: https://doi.org/10.1109/ACCESS.2017.2685629. (Cited on page 69.)

[162] Hossain Shahriar and Mohammad Zulkernine. Mutation-based testing of buffer overflow vulnerabilities. In *2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 979–984. IEEE, 2008. doi: https://doi.org/10.1109/COMPSAC.2008.123. (Cited on page 16.)

[163] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. Comparing and combining test-suite reduction and regression test selection. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 237–247. ACM, 2015. doi: https://doi.org/10.1145/2786805.2786878. (Cited on page 104.)

[164] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. Evaluating test-suite reduction in real software evolution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 84–94. ACM, 2018. doi: https://doi.org/10.1145/3213846.3213875. (Cited on page 104.)

[165] Francisco Carlos M. Souza, Mike Papadakis, Yves Le Traon, and Márcio E. Delamaro. Strong mutation-based test data generation using hill climbing. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, pages 45–54. ACM, 2016. doi: https://doi.org/10.1145/2897010.2897012. (Cited on page 106.)

[166] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB®-Standard*. dpunkt.verlag, 2019. ISBN 9783960885023. (Cited on page 12, 13, 14, and 15.)

[167] Ofer Strichman and Benny Godlin. *Regression Verification - A Practical Way to Verify Programs*, volume 4171 of *LNCS*, pages 496–501. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-69149-5. doi: https://doi.org/10.1007/978-3-540-69149-5_54. (Cited on page 106.)

[168] Abu-Bakr Taha, Stephen M. Thebaut, and Sying-Syang Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In *[1989] Proceedings of the Thirteenth Annual International Computer Software Applications Conference*, pages 527–534. IEEE, 1989. doi: https://doi.org/10.1109/CMPSAC.1989.65142. (Cited on page 104.)

[169] Kunal Taneja and Tao Xie. Diffgen: Automated regression unit-test generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 407–410. IEEE, 2008. doi: https://doi.org/10.1109/ASE.2008.60. (Cited on page 69, 76, and 106.)

[170] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration coverage in the analysis of large-scale system software. *SIGOPS Oper. Syst. Rev.*, 45:10–14, 2012. doi: https://doi.org/10.1145/2094091.2094095. (Cited on page 136.)

[171] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Static analysis of variability in system software: The 90,000 #ifdefs issue. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, pages 421–432. USENIX Association, 2014. (Cited on page 136.)

[172] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.*, 47:6:1–6:45, 2014. doi: https://doi.org/10.1145/2580950. (Cited on page 110, 112, 138, and 141.)

[173] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract features in feature modeling. In *15th International Software Product Line Conference*, pages 191–200. IEEE, 2011. doi: https://doi.org/10.1109/SPLC.2011.53. (Cited on page 34.)

[174] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammadreza Mousavi, and Ina Schaefer. A Classification of Product Sampling for Software Product Lines. In *SPLC*, volume 1, page 1–13. ACM, 2018. doi: https://doi.org/10.1145/3233027.3233035. (Cited on page 135.)

[175] Till Hänisch Volker P. Andelfinger. *Industrie 4.0*. Springer Fachmedien Wiesbaden, 2017. ISBN 3658155566. (Cited on page 7.)

[176] Raphael Wagner, Benjamin Schleich, Benjamin Haefner, Andreas Kuhnle, Sandro Wartzack, and Gisela Lanza. Challenges and potentials of digital twins and industry 4.0 in product design and production for high performance products. *Procedia CIRP*, 84:88–93, 2019. doi: https://doi.org/10.1016/j.procir.2019.04.219. (Cited on page 10.)

[177] Kaiyuan Wang, Chenguang Zhu, Ahmet Celik, Jongwook Kim, Don Batory, and Milos Gligoric. Towards refactoring-aware regression test selection. In *Proceedings of the 40th International Conference on Software Engineering*, pages 233–244. ACM, 2018. doi: https://doi.org/10.1145/3180155.3180254. (Cited on page 66 and 104.)

[178] Xiaolin Wang and Hongwei Zeng. History-based dynamic test case prioritization for requirement properties in regression testing. In *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery*, pages 41–47. IEEE, 2016. doi: https://doi.org/10.1145/2896941.2896949. (Cited on page 105.)

[179] Weichen E. Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. Effect of test set minimization on fault detection effectiveness. In *1995 17th International Conference on Software Engineering*, pages 41–41. ACM, 1995. doi: https://doi.org/10.1145/225014.225018. (Cited on page 105.)

[180] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 46, pages 283–294. ACM, 2011. doi: https://doi.org/10.1145/1993498.1993532. (Cited on page 105.)

[181] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22:67–120, 2012. doi: https://doi.org/10.1002/stv.430. (Cited on page 3, 26, 27, 28, 29, 69, 71, 74, 76, 101, 103, and 140.)

[182] Michał Zalewski. Technical "whitepaper" for afl-fuzz. Technical report, 2006. URL `https://lcamtuf.coredump.cx/afl/technical_details.txt`. (Cited on page 65.)

A

SUBJECT SYSTEMS

Table A.1: Chapter 3 Subject Systems

Subject System	File	Function
busybox-1.22.0	echo-1.i	main
busybox-1.22.0	echo-2.i	main
combinations	pals_lcr.3.ufo.BOUNDED-6.pals+Problem12_label00.c	main
combinations	pals_lcr.3.ufo.BOUNDED-6.pals+Problem12_label01.c	main
combinations	pals_lcr.3.ufo.BOUNDED-6.pals+Problem12_label02.c	main
combinations	pals_lcr.3.ufo.BOUNDED-6.pals+Problem12_label04.c	main
combinations	pals_lcr.3.ufo.BOUNDED-6.pals+Problem12_label05.c	main
combinations	pals_lcr.3.ufo.BOUNDED-6.pals+Problem12_label08.c	main
combinations	pals_lcr.3.ufo.BOUNDED-6.pals+Problem12_label09.c	main
combinations	pals_lcr.3.ufo.UNBOUNDED.pals+Problem12_label00.c	main
combinations	pals_lcr.3.ufo.UNBOUNDED.pals+Problem12_label01.c	main
combinations	pals_lcr.3.ufo.UNBOUNDED.pals+Problem12_label02.c	main
combinations	pals_lcr.3.ufo.UNBOUNDED.pals+Problem12_label03.c	main
combinations	pals_lcr.3.ufo.UNBOUNDED.pals+Problem12_label04.c	main
combinations	pals_lcr.3.ufo.UNBOUNDED.pals+Problem12_label05.c	main
combinations	pals_lcr.3.ufo.UNBOUNDED.pals+Problem12_label06.c	main
combinations	pals_lcr.3.ufo.UNBOUNDED.pals+Problem12_label07.c	main
combinations	pals_lcr.3.ufo.UNBOUNDED.pals+Problem12_label08.c	main
combinations	pals_lcr.3.ufo.UNBOUNDED.pals+Problem12_label09.c	main
combinations	pals_lcr.5.1.ufo.BOUNDED-10.pals+Problem12_label09.c	main
combinations	pals_lcr.5.ufo.BOUNDED-10.pals+Problem12_label05.c	main
combinations	pals_lcr.6_overflow.ufo.UNBOUNDED.pals+Problem12_label04.c	main
eca-rers2012	Problem10_label00.c	main
eca-rers2012	Problem11_label00.c	main
eca-rers2012	Problem12_label00.c	main
eca-rers2012	Problem13_label00.c	main
eca-rers2012	Problem15_label00.c	main
eca-rers2012	Problem16_label00.c	main
eca-rers2012	Problem17_label00.c	main
eca-rers2012	Problem18_label00.c	main
eca-rers2018	Problem10.c	main
float-newlib	double_req_bl_0870a.c	main
float-newlib	double_req_bl_0874.c	main
float-newlib	double_req_bl_0876.c	main
float-newlib	double_req_bl_0882.c	main
float-newlib	double_req_bl_0883.c	main
float-newlib	float_req_bl_0880.c	main
float-newlib	float_req_bl_0881.c	main
ldv-linux-3.4-simple	2_1_cilled_ok_nondet_linux-3.4-32_1-drivers–hwmon–gpio-fan.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.i	main
ldv-linux-3.4-simple	32_1_cilled_ok_nondet_linux-3.4-32_1-drivers–message–i2o–i2o_scsi.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.i	main

ldv-linux-3.4-simple	43_1a_cilled_ok_nondet_linux-43_1a-drivers–hwmon–gpio-fan.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.i	main
ldv-linux-3.4-simple	43_1a_cilled_ok_nondet_linux-43_1a-drivers–message–i2o–i2o_scsi.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.i	main
ldv-linux-3.4-simple	43_1a_cilled_ok_nondet_linux-43_1a-drivers–mmc–host–sdricoh_cs.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.i	main
loop-invgen	apache-get-tag.i.p+lhb-reducer.c	main
loops	s3.i	main
ntdrivers-simplified	diskperf_simpl1.cil.c	main
openssl	s3_clnt.blast.01.i.cil-1.c	main
openssl	s3_clnt.blast.01.i.cil-2.c	main
openssl	s3_clnt.blast.02.i.cil-1.c	main
openssl	s3_clnt.blast.02.i.cil-2.c	main
openssl	s3_clnt.blast.03.i.cil-1.c	main
openssl	s3_clnt.blast.03.i.cil-2.c	main
openssl	s3_clnt.blast.04.i.cil-1.c	main
openssl	s3_clnt.blast.04.i.cil-2.c	main
psyco	psyco_abp_1-2.c	main
seq-mthreaded	pals_STARTPALS_ActiveStandby.1.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_STARTPALS_ActiveStandby.4_1.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_STARTPALS_ActiveStandby.4_2.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_STARTPALS_ActiveStandby.5.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_STARTPALS_ActiveStandby.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_STARTPALS_Triplicated.1.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_STARTPALS_Triplicated.2.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_STARTPALS_Triplicated.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_floodmax.3.1.ufo.BOUNDED-6.pals.c	main
seq-mthreaded	pals_floodmax.3.2.ufo.BOUNDED-6.pals.c	main
seq-mthreaded	pals_floodmax.3.2.ufo.UNBOUNDED.pals.c	main
seq-mthreaded	pals_floodmax.3.3.ufo.BOUNDED-6.pals.c	main
seq-mthreaded	pals_floodmax.3.4.ufo.BOUNDED-6.pals.c	main
seq-mthreaded	pals_floodmax.3.ufo.BOUNDED-6.pals.c	main
seq-mthreaded	pals_floodmax.4.1.ufo.BOUNDED-8.pals.c	main
seq-mthreaded	pals_floodmax.4.3.ufo.BOUNDED-8.pals.c	main
seq-mthreaded	pals_floodmax.4.4.ufo.BOUNDED-8.pals.c	main
seq-mthreaded	pals_floodmax.4.ufo.BOUNDED-8.pals.c	main
seq-mthreaded	pals_floodmax.5.1.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_floodmax.5.2.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_floodmax.5.3.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_floodmax.5.4.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_floodmax.5.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_lcr-var-start-time.6.1.ufo.BOUNDED-12.pals.c	main
seq-mthreaded	pals_lcr-var-start-time.6.2.ufo.BOUNDED-12.pals.c	main
seq-mthreaded	pals_lcr-var-start-time.6.ufo.BOUNDED-12.pals.c	main
seq-mthreaded	pals_lcr.7.1.ufo.BOUNDED-14.pals.c	main
seq-mthreaded	pals_lcr.7.ufo.BOUNDED-14.pals.c	main
seq-mthreaded	pals_lcr.8.1.ufo.BOUNDED-16.pals.c	main
seq-mthreaded	pals_lcr.8.ufo.BOUNDED-16.pals.c	main
seq-mthreaded	pals_opt-floodmax.3.1.ufo.BOUNDED-6.pals.c	main
seq-mthreaded	pals_opt-floodmax.3.2.ufo.BOUNDED-6.pals.c	main
seq-mthreaded	pals_opt-floodmax.3.3.ufo.BOUNDED-6.pals.c	main
seq-mthreaded	pals_opt-floodmax.3.4.ufo.BOUNDED-6.pals.c	main
seq-mthreaded	pals_opt-floodmax.3.ufo.BOUNDED-6.pals.c	main
seq-mthreaded	pals_opt-floodmax.4.1.ufo.BOUNDED-8.pals.c	main
seq-mthreaded	pals_opt-floodmax.4.2.ufo.BOUNDED-8.pals.c	main
seq-mthreaded	pals_opt-floodmax.4.3.ufo.BOUNDED-8.pals.c	main
seq-mthreaded	pals_opt-floodmax.4.4.ufo.BOUNDED-8.pals.c	main

seq-mthreaded	pals_opt-floodmax.4.ufo.BOUNDED-8.pals.c	main
seq-mthreaded	pals_opt-floodmax.5.1.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_opt-floodmax.5.2.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_opt-floodmax.5.3.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_opt-floodmax.5.4.ufo.BOUNDED-10.pals.c	main
seq-mthreaded	pals_opt-floodmax.5.ufo.BOUNDED-10.pals.c	main

Table A.2: Chapter 4 Subject Systems

Subject System	File	Function
Betaflight	initialisation.c	targetBusInit
Betaflight	serial.c	serialWrite
Betaflight	serialx.c	frameStatus
Betaflight	serialx.c	readRawRC
Betaflight	sumh.c	sumhFrameStatus
Betaflight	sumh.c	sumhReadRawRC
netdata	appconfig.c	config_get
netdata	appconfig.c	config_get_boolean
netdata	appconfig.c	config_get_number
netdata	appconfig.c	config_set_boolean
netdata	appconfig.c	load_config
netdata	daemon.c	sig_handler
wrk	stats	stats_record

Table A.3: Chapter 5 Subject Systems

Subject System	File	Function
-	calculate.c	calculate
-	lcm.c	lcm
-	rand_int.c	rand_int
Busybox	-	is_method1_triggered
Busybox	passwd.c	passwd_main
Busybox	deluser.c	deluser_main.c
VIM	-	minimized_get_varp
VIM	gui.c	gui_get_base_height
VIM	gui.c	gui_init_check
VIM	options.c	insecure_flag

B

MUTATION OPERATORS

Table B.1: Chapter 4 Mutation Operators

Mutation Operator	Description
CGCR	Constant replacement using global constants
CLCR	Constant replacement using local constants
CGSR	Scalar variable replacement using global constants
CLSR	Scalar variable replacement using local constants
CRCR	Integer constants replacement
OAAA	Arithmetic Operator Assignment
OAAN	Arithmetic operator replacement
OABA	arithmetic assignment by bitwise assignment
OABN	arithmetic operator by bitwise operator
OAEA	arithmetic assignment by plain assignment
OALN	arithmetic operator by logical operator
OARN	arithmetic operator by relational operator
OASA	arithmetic assignment by shift assignment
OASN	Replacement of an arithmetic operator with shift operator
OBAA	Bitwise assignment by arithmetic assignment
OBAN	Bitwise operator by arithmetic assignment
OBBA	Bitwise assignment mutation
OBBN	Bitwise logical replacement
OBEA	Bitwise assignment by plain assignment
OBLN	Bitwise operator by logical operator
OBNG	Bitwise negation
OBRN	Bitwise operator by relational operator
OBSA	Bitwise assignment by shift assignment
OBSN	Bitwise operator by shift operator
OCNG	Logical context negation
OCOR	Cast operator by cast operator
OEAA	Plain assignment by arithmetic assignment
OEBA	Replacement of a plain assignment with bitwise assignment
OESA	Plain assignment by shift assignment
OPPO	Increment by decrement replacement
OMMO	Decrement by increment replacement
OLAN	Logical operator by arithmetic operator
OIPM	Indirection operator precedence mutation
OLBN	Logical operator by bitwise operator
OLLN	Logical operator replacement
OLNG	Logical negation
OLRN	Logical operator by relational operator
OLSN	Logical operator by shift operator
ORAN	Relational operator by arithmetic operator
ORBN	Relational operator by bitwise operator
ORLN	Relational operator by Logical operator
ORRN	Replacement of a relational operator with other relational operator
ORSN	Relational operator by shift operator

OSAA	Shift assignment by arithmetic assignment
OSAN	Shift operator by arithmetic operator
OSBA	Shift assignment by bitwise assignment
OSBN	Shift operator by bitwise operator
OSEA	Shift assignment by plain assignment
OSLN	Shift operator by logical operator
OSRN	Shift operator by relational operator
OSSA	Shift assignment mutation
OSSN	Shift operator mutation
VGAR	Mutate array references using global array references
VGPR	Mutate pointer references using global pointer references
VGSR	Mutate scalar references using global scalar references
VGTR	Mutate structure references using global structure references
VLAR	Mutate array references using local array references
VLPR	Mutate pointer references using local pointer references
VLSR	Mutate scalar references using local scalar references
VLTR	Mutate structure references using only local structure references
VSCR	Structure component replacement
VTWD	Changes the value of scalar variables for the value of its predecessor / successor

Table B.2: Chapter 5 Mutation Operators

Mutation Operator	Description
ABS	ABS Insertion
CRCR	Integer constants replacement
OAAA	Arithmetic Operator Assignment
OAAN	Arithmetic operator replacement
OBBA	Bitwise assignment mutation
OCNG	Logical context negation
OIDO	Decrement/Increment Replacement
OLLN	Logical operator replacement
OLNG	Logical negation
ORRN	Replacement of a relational operator with other relational operator
FVDL	Feature Variable Replacement
OFCNG	Presence Condition Negation Replacement
OFLLN	And/Or Presence Condition Replacement
VFXFYR	Feature Variable Replacement

C

AUTHOR'S PUBLICATIONS

- Sebastian Ruland and Malte Lochau. On the interaction between test-suite reduction and regression-test selection strategies. arXiv, 2022.
  doi: https://doi.org/10.48550/arXiv.2207.12733.
- Vladimir Kutscher, Sebastian Ruland, Patrick Müller, Nathan Wasser, Malte Lochau, Reiner Anderl, Andy Schürr, Mira Mezini, and Reiner Hähnle. Towards a circular economy of industrial software. Procedia CIRP, 90:37–42, 2020.
  doi: https://doi.org/10.1016/j.procir.2020.01.133.
- Sebastian Ruland, Géza Kulcsár, Erhan Leblebici, Sven Peldszus, and Malte Lochau. On controlling the attack surface of object-oriented refactorings. In Software Engineering 2020, volume P-300, pages 89–90. Gesellschaft für Informatik e.V., 2020.
  doi: https://doi.org/10.18420/SE2020_26.
- Sebastian Ruland, Malte Lochau, Oliver Fehse, and Andy Schürr. CPA/Tiger-MGP: test-goal set partitioning for efficient multi-goal test-suite generation. International Journal on Software Tools for Technology Transfer, pages 1–4, 2020.
  doi: https://doi.org/10.1007/s10009-020-00574-z.
- Sebastian Ruland, Malte Lochau, and Marie-Christine Jakobs. HybridTiger: Hybrid Model Checking and Domination-based Partitioning for Efficient Multi-Goal Test-Suite Generation (Competition Contribution). In Fundamental Approaches to Software Engineering, volume 12076, pages 520–524. Springer International Publishing, 2020. doi: https://doi.org/10.1007/978-3-030-45234-6_26
- Dennis Reuling, Udo Kelter, Sebastian Ruland, and Malte Lochau. Simpose - configurable n-way program merging strategies for superimposition-based analysis of variant-rich software. In 34th IEEE/ACM International Conference on Automated Software Engineering, pages 1134–1137. IEEE Press, 2019. doi: https://doi.org/10.1109/ASE.2019.00120.
- Sebastian Ruland, Géza Kulcsár, Erhan Leblebici, Sven Peldszus, and Malte Lochau. Controlling the attack surface of object-oriented refactorings. In Fundamental Approaches to Software Engineering, pages 38–55. Springer International Publishing, 2018.
  doi: https://doi.org/10.1007/978-3-319-89363-1_3.
- Sebastian Ruland, Lars Luthmann, Johannes Bürdek, Sascha Lity, Thomas Thüm, Malte Lochau, and Márcio Ribeiro. Measuring Effectiveness of Sample-Based Product-line Testing. In Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, pages 119–133. ACM, 2018.
  doi: https://doi.org/10.1145/3278122.3278130.