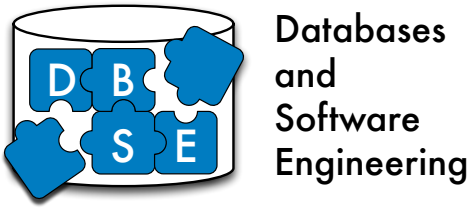
University of Magdeburg

School of Computer Science



Dissertation

Variational Debugging: Understanding Differences among Executions

Author:

Jens Meinicke

7. Januar 2019

Reviewers:

Prof. Gunter Saake (University of Magdeburg, Germany)

Prof. Christian Kästner (Carnegie Mellon University, PA, USA)

Prof. Xiangyu Zhang (Purdue University, IN, USA)

Meinicke, Jens:
Variational Debugging: Understanding Differences among Executions
Dissertation, University of Magdeburg, 2019.

Variational Debugging: Understanding Differences among Executions

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von M.Sc. Jens Meinicke

geb. am 01.11.1988 in Halle (Saale)

Gutachterinnen/Gutachter:

Prof. Gunter Saake (University of Magdeburg, Germany)

Prof. Christian Kästner (Carnegie Mellon University, PA, USA)

Prof. Xiangyu Zhang (Purdue University, IN, USA)

Magdeburg, den 7.1.2019

Abstract

Interactions among multiple program inputs or options can lead to undesired or wrong behavior, such as system crashes and security vulnerabilities. This is especially challenging for large numbers of inputs such as for highly configurable systems, as the number of configurations to test grows exponentially with the number of boolean options. A lot of research has focused on systematically covering this configuration space, however there is little knowledge on how inputs interact as they can only be observed from their effects (e.g., from bug reports). A better understanding of interactions is needed to improve quality assurance.

In this thesis, we developed a dynamic analysis based on variational execution, that monitors data and control flow interactions among all options simultaneously. We analyzed the program traces of multiple medium sized highly-configurable systems to characterize and identify where and how interactions occur. We found that the essential configuration complexity (i.e., the degree of interactions occurring during executions) is indeed much lower than the combinatorial explosion, but that the pattern of how options interact are more nuanced than what state of the art analysis techniques exploit.

Interaction characteristics can inform analysis techniques, however, understanding of a specific interaction (e.g., a system crash), is challenging as this requires understanding how multiple inputs interact with each other to cause this undesired behavior while the program succeeds otherwise. Debugging such faults requires understanding the individual effects of inputs and how they interact to cause the fault. Contrasting traces of failing and the succeeding executions can reveal the interactions in their differences, and thus the information needed to understand the fault. We propose to align the execution traces of all configurations to discover and explain interactions. As complete traces are too large to be used for debugging, we present variational traces that concisely represent the differences on data and control flow caused by interactions. We again use variational executions to scale the generation of variational traces to exponential configuration spaces. To enable programmers interacting with such variational traces, we provide our debugging tool Varviz. We have shown that variational traces improve the performance of debugging variability faults by a factor of two compared to standard debuggers.

Zusammenfassung

Interaktionen von mehreren Programmparametern oder Optionen können zu unerwünschten oder falschen Verhalten, wie zum Beispiel Systemabstürzen oder Sicherheitslücken, führen. Das ist vor allem für eine große Anzahl von Parametern herausfordernd, da die Menge der zu testenden Konfigurationen exponentiell mit der Zahl der Parameter steigt. Viel Forschung wurde betrieben, um diesen Konfigurationsraum systematisch abzudecken, allerdings ist wenig darüber bekannt wie Optionen tatsächlich interagieren da Interaktionen meist nur durch ihren Effekt beobachtbar sind (z.B. durch Fehlerberichte). Ein besseres Verständnis von Interaktionen wird benötigt um die Qualitätssicherung von Software zu verbessern.

In dieser Dissertation, haben wir einen dynamischen Ansatz entwickelt der auf Variational Execution basiert. Dieser Ansatz beobachtet Interaktionen in Daten und Kontrollfluss zwischen allen Optionen simultan. Wir haben Programmausführungen von mehreren mittelgroßen Programmen analysiert um zu charakterisieren und zu identifizieren wie und wo Interaktionen auftreten. Wir haben herausgefunden, dass die essentielle Konfigurationskomplexität (d.h. der Grad in dem Interaktionen während der Ausführung auftreten) tatsächlich wesentlich niedriger ist als die kombinatorische Explosion. Allerdings sind die Muster in denen Interaktionen auftreten nuancierter als aktuelle Techniken derzeit ausnutzen.

Zwar können Charakteristiken über Interaktionen Analysen verbessern, allerdings bleibt das Verstehen einer bestimmten Interaktion herausfordernd da man verstehen muss wie mehrere Optionen miteinander interagieren um dieses unerwünschte Verhalten zu verursachen während das Programm sich sonst normal verhält. Um solche Fehler zu debuggen muss man die Effekte einzelner Optionen und deren Interaktionen verstehen. Vergleichen von fehlerhaften und korrekten Ausführungen kann die Interaktionen anhand der Unterschiede aufzeigen, und somit die Informationen die zum Verständnis des Fehlers benötigt sind. Wir schlagen vor die Ausführungen aller Konfigurationen zu vergleichen um Interaktionen zu finden und zu erklären. Da komplette Ausführungen zu lang sind um für das Debuggen nutzbar zu sein, präsentieren wir Variational Traces, welche die durch Interaktionen verursachten Unterschiede in Daten und Kontrollfluss kompakt repräsentieren. Wir nutzen Variational Execution um die Generierung von Variational Traces für exponentiell große Konfigurationsräume zu skalieren. Wir stellen unser Debugging-Werkzeug Varviz zur Verfügung um es Programmierern zu erlauben Variational Traces für das Debuggen von Interaktionen zu nutzen. Wir haben gezeigt, dass Variational Traces die Leistung beim Debuggen von Variabilitätsfehlern um einen Faktor von Zwei gegenüber eines Standard-Debuggers verbessern.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

Most of today's software provides some sort of configuration, such as most end-user programs, such as browsers, safety critical systems, such as cars, but also security sensitive software, such as SSL libraries and databases. Software variability allows to customize the behavior of a program regarding the requirements of its users [Apel et al., 2013a, Clements and Northrop, 2001, Pohl et al., 2005]. Variability enables a program to be specialized for thousands of different users with customized configurations. However, this flexibility comes at the cost of complexity. Each additional boolean option doubles the number of configurations, if there are no dependencies among the options. Thus, the number of configurations grows exponentially, causing the so-called *configuration space explosion* (a.k.a. combinatorial explosion). Analyzing each configuration individually is usually too time consuming, impractical or even impossible [Halin et al., 2017, Medeiros et al., 2015, Thüm et al., 2014]. However, testing each feature in isolations is not sufficient either, as fault are often caused by unexpected interactions among them, known as the *feature interaction problem* [Abal et al., 2018, Apel et al., 2013b, Bruns, 2005, Calder et al., 2003a,b, Garvin and Cohen, 2011, Nhlabatsi et al., 2008, Zave, 2009]. Thus, in practice, usually only one or few configurations are tested and detecting faults is left to users [Carmo Machado et al., 2014, Greiler et al., 2012, Medeiros et al., 2015].

Feature interactions are hard to find and can have severe consequences. As feature interactions are a general problem, they were reported in many different domains [Abal et al., 2018, Crespo et al., 2007, Donaldson and Calder, 2012, Georgiev et al., 2012, Jayaraman et al., 2007, Juarez Dominguez, 2012, Nhlabatsi et al., 2008, Weiss et al., 2007]. Faults due to feature interactions can lead to system crashes, leakage of sensitive data, server outages and severe problems in safety critical systems. Detecting feature interactions upfront seems impossible due to the huge configuration spaces of these systems, especially when more than two options are interacting.

There has been a lot of research providing specialized analyses and tools to detect faults caused by variability and feature interactions [Meinicke et al., 2014, Thüm et al., 2014]. These analyzes are based on common assumptions, that only few

features interact and that most interaction faults can be detected by covering interactions among three features [Abal et al., 2018, Garvin and Cohen, 2011]. However, these assumptions are based on bug reports and are biased towards frequently used configurations [Abal et al., 2018]. An exhaustive analysis of interactions is expensive and can only detect interactions by their observable effects [Halin et al., 2018] However, higher degree interactions exist: for example Reisner et al. [2010] found that 7 of 30 options interact, and Nguyen et al. [2014b] detected interactions among 16 out of 50 plugins. These high degree interactions can potentially cause unexpected behaviors, that are unlikely to be detected with approaches that only cover interactions among few options.

Our goal is to improve the understanding about feature interactions by observing them directly when they occur at runtime. A common approach to understand the effects of options is to compare the executions for different configurations [Sumner and Zhang, 2013, Zeller, 2002]. Understanding the differences among the executions not only requires comparing the results of the program but also comparing the execution traces and program states. Comparing the executions comes with challenges for *scalability*, *trace alignment* and *non-determinism* [Kwon et al., 2016, Sumner and Zhang, 2010, 2013, Zeller, 2002]. For n independent options this requires the program to be executed 2^n times which can only scale to low numbers of options. Thus, recent approaches only compare few (usually two) executions at a time [Kwon et al., 2016, Sumner and Zhang, 2010, 2013, Zeller, 2002] or only observe interactions on code coverage [Reisner et al., 2010] and reachability [Lillack et al., 2017]. As we aim to understand feature interactions, we need to compare the executions of *all* configurations and observe differences on data and control-flow caused by feature interactions.

We use the approach of variational execution (a.k.a. variability-aware execution and faceted execution) to compare all, exponentially many, executions [Austin et al., 2013, Kästner et al., 2012b, Meinicke, 2014, Meinicke et al., 2016, Nguyen et al., 2014b, Schmitz et al., 2016, 2018, Wong et al., 2018b, Yang et al., 2016]. Variational execution is an approach that can execute an exponentially large number of configurations, mostly efficiently, by aggressively sharing redundancies in executions and in data. The basis of this thesis is the realization that this sharing ability of variational execution, is equivalent to an alignment of all executions. This allows us to compare all executions, and thus, observe feature interactions directly.

The goal of this thesis is to gain an understanding of feature interactions by observing them directly during runtime using variational execution. Based on this, we help developers and researchers dealing with feature interactions by providing common characteristics and a novel approach of variational traces that allows to inspect and debug feature interactions.

1.1 Contribution

This thesis has two main contributions: First, we analyze how options interact at runtime to gain a better understanding of common characteristics of interactions. These insights help to understand why certain analysis do not scale to analyze configurable systems and why they cannot detect all interactions. It also helps researchers to develop new analyses by being aware of which interaction characteristics can be exploited.

Second, we developed support that helps users to understand and debug interactions among options. Our approach helps to understand how options interact, which allows developers to debug configurable systems with huge configuration spaces and interactions among many options.

Interaction characteristics

Most assumptions on feature interactions come from external observations, such as bug reports [Abal et al., 2018]. To confirm and complete these assumptions we characterize how options interact in highly configurable systems by observing how options interact in control and data flow during runtime:

1. We developed a *dynamic analysis* for Java that tracks interactions on data and control flow during execution. Our analysis enables a fine-grained observation of interactions compared to state-of-the-art approaches which have scalability issues and can only observe interactions on code coverage [Reisner et al., 2010] and reachability [Lillack et al., 2017].

2. We developed three measures that *characterize essential configuration complexity*, measuring how options interact within an execution. These measures help us to characterize how options interact which allows us to make general statements of how options usually interact. These characteristics explain scalability of existing analyses and can lead to more efficient analyses that exploit them.

3. State-of-the-art analyses for configurable systems aim to exploit interaction characteristics to scale to exponential configuration spaces. We designed five *interaction benchmarks* to study how well these analyses scale for different interaction characteristics. Our results expose why certain approaches do not scale for certain kinds of interactions.

4. We measured the configuration complexity for medium-sized systems, finding that essential configuration complexity is low enough to enable configuration-complete analyses.

5. We discuss common characteristics of interactions, providing more nuanced variants of current assumptions, which can, among others, encourage more efficient analyses for exponential configuration spaces.

Variational Debugging

Debugging is a difficult and time-consuming task in software development. The additional dimension of variability increases the difficulty and thus time for debugging [Melo et al., 2016]. In the second part of this thesis, we aim to help developers debugging faults in configurable systems.

1. We developed the concept of a *variational trace* that compactly represents differences among the interactions of all configurations.

2. Generating variational traces requires logging and aligning the executions for all configurations. We provide a *baseline implementation* that emphasizes challenges of generating variational traces, such as memory consumption and executions time.

3. We implemented mechanisms based on variational execution [Meinicke, 2014, Meinicke et al., 2016] for *efficient trace alignment* and a *dynamic analysis to trace only relevant data* which avoid memory explosion for a potentially exponential number of configurations.

4. We provide an Eclipse plug-in *Varviz* to visualize and interact with variational traces to allow developers debug interaction faults.

5. We performed a qualitative *user study* which shows that participants using variational traces outperform participants using a standard debugger. The study also shows that comparative approaches [Sumner and Zhang, 2013, Zeller, 2002] actually help with debugging tasks.

1.2 Broader Impact

Advances in variational execution made in this thesis will change how configurable systems are tested. These advances improve quality assurance of most of today's end user software as well as security sensitive software such as databases and SSL libraries. Variational execution provides a simple, automatic and efficient way to detect feature interactions, in contrast to more complicated techniques such as model checking. Thus, variational execution is applicable by researchers and practitioners, but also hobbyists, while improving the overall software quality and reducing the cost of quality assurance at the same time.

We further refine our understanding of feature interactions. Our new insights will help designing better quality assurance strategies that are aware of how options interact in software. These insights enable us to explain why current analyses scale for certain feature interactions but do not for others. With our new understanding on feature interactions, we help building and designing configurable software that is easier to analyze by being aware of which types of interactions are challenging to analyze or are supported by existing analyses.

With variational traces, we provide the first opportunity to directly observe feature interactions when they occur during runtime; in contrast to black box approaches

that only observe their effects (e.g., interaction faults and assertions). The ability to observe feature interactions directly enables future researchers to study them in more detail. Variational traces further help developers to debug interaction faults, which are notoriously hard to understand with standard tools. Beyond the usefulness of variational traces, our study shows for the first time, that comparative approaches (i.e., contrasting executions) actually help developers understanding and debugging faults.

Variational execution and variational debugging have promising applications beyond configurable systems when exploring many variations of a program. In the area of mutation testing, exploring many mutations and their combinations can help finding good mutations. As variational execution explores all combinations of mutants we can efficiently detect higher-order mutations, compared to existing search-based approaches. Similarly, variational execution can be used to explore many patch candidates for search-based automatic program repair. Variational execution explores combinations of patches, which helps finding multi-line patches which are challenging to detect with current approaches.

The contributions of this thesis lead to better software quality by detecting faults easier, by guiding better analysis in the future, and by informing the system design to avoid potentially-difficult-to-analyze interactions. This leads to improved reliability, safety, security while reducing the development cost.

1.3 Structure of the Thesis

In Chapter 2 (*Feature Interactions*) we introduce the background on feature interactions, which gives a detailed motivation for this thesis.

In Chapter 3 (*Variational Execution*) we introduce the variational execution which is the basis for this thesis. This way we enable readers to understand the benefits of variational execution as dynamic analysis to observe feature interactions.

In Chapter 4 (*Measuring Interactions in Highly Configurable Systems*) we use variational execution to measure feature interactions during runtime. In this chapter, we present the contributions regarding *interaction characteristics*.

In Chapter 5 (*Understanding Interactions in Highly-Configurable Systems with Variational Traces*) we develop the concept of variational traces to help developers debug feature interactions. In this chapter we present the contributions regarding *variational debugging*.

In Chapter 6 (*Conclusion*) we summarize the contributions of our work and discuss future directions.

2. Feature Interactions

Variability allows developers and users to customize programs to meet special need, such as functional or performance requirements [Apel et al., 2013a, Clements and Northrop, 2001]. However, variability comes with the cost of increased complexity which hinders *code comprehension*, *program analysis* and *debugging*. Variability is often implemented in terms of features. A *feature* is a user visible aspect of the software [Apel et al., 2013a]. Variability is implemented by enabling or disabling these features. Thus, a *configuration* is a set of enabled and disabled features. Configurations are used to derive a product that contains the functionalities of the enabled features [Apel et al., 2013a, Clements and Northrop, 2001]. This process allows customizability which is used to meet user needs, hardware requirements, or to optimize performance. Prominent examples off highly configurable systems are operating systems (e.g., Linux), and browsers (e.g., Chrome).

Analysis of highly configurable systems is problematic as tests may succeeded in one configuration but fail in others [Cohen et al., 2007, Medeiros et al., 2016, Nie and Leung, 2011]. Such faults are often caused by *feature interactions* [Abal et al., 2018, Apel et al., 2013b, Calder et al., 2003a, Garvin and Cohen, 2011]. A *feature interaction* is a situation in which two or more features affect each other's behavior [Bruns, 2005, Calder et al., 2003b, Nhlabatsi et al., 2008, Zave, 2009]. Feature interactions became prominent in the 90s when studying telecommunication systems [Calder et al., 2003b]. Research on feature interactions originates from requirements engineering and feature-driven development where systems are compositions of features that are however not considered optional [Bruns, 2005, Calder et al., 2003b, Palmer and Felsing, 2001, Zave, 2009]. Software product line engineering continued this research but considered features as optional aspects [Apel et al., 2013a, Clements and Northrop, 2001, Pohl et al., 2005]. Feature interactions are a common problem of configurable systems as it is impossible to reason about all possible interactions manually. Thus, feature interactions can lead to undesired behaviors, such as system crashes [Abal et al., 2018, Garvin and Cohen, 2011, Kuhn et al., 2004, Nhlabatsi et al., 2008]. However, feature interactions are hard to detect as they only appear in specific configurations. The additional layer of variability increases the code complexity which makes it harder to understand the program [Melo et al., 2016].

Faults that appear only in a certain of configurations are called *variability faults* [Abal et al., 2018]. *Feature interaction faults* are a subset of variability faults that are caused by certain combinations of two or more options (e.g., $A \wedge \neg B$ or $A \wedge B \wedge C$). The *interaction degree* defines the number of options that interact. Feature interaction with high interaction degree are harder to detect, but considered to be less common on practice [Abal et al., 2018, Garvin and Cohen, 2011]. Feature interaction faults are hard to detect as they only appear in few configurations [Kuhn et al., 2004, Nhlabatsi et al., 2008, Thüm et al., 2014] and hard to understand as they often require reasoning about the effects of multiple features [Melo et al., 2016].

In practice, the configuration space is rarely explored systematically [Halin et al., 2017]. Instead ad-hoc testing of single configurations is still common and testing configurations is left to users [Carmo Machado et al., 2014, Greiler et al., 2012, Medeiros et al., 2015]. At best combinatorial interaction testing is used which checks for all combination of only few options [Nie and Leung, 2011]. Thus, variability faults are often only discovered by users [Greiler et al., 2012, Medeiros et al., 2015].

There is only little knowledge about feature interactions as they are only observed in bug reports but usually not by systematically exploring the complete configuration space (e.g., by testing all configurations with a brute-force approach) [Abal et al., 2018, Halin et al., 2017]. Also, feature interactions may not manifest in observable faults, but in internal behavioral differences. Such types of feature iterations are difficult to detect as they may not be observable from the outside.

configurable systems	debugging	information-flow
option/feature	input	privacy-/security-/confidentially-level
configuration	set of inputs	set of private variables

Table 2.1: Translation of terminologies from configurable systems to debugging and information flow.

The challenges of feature interactions in configurable systems can be translated to the fields of debugging and information flow. We summarize the translations of terminologies in Table 2.1. For the sake of simplicity, we use the terminology of configurable systems. The challenges and our solutions are similar. We use the established term feature interaction to refer to interactions among options.

2.1 Quality assurance for highly-configurable systems

Boolean configuration options are a special type of inputs that, in contrast to arbitrary inputs (e.g., Strings), represent a finite set of possible configurations. Even though the number of configurations can be exponentially large to the number of independent options, it is usually not necessary to analyze the program for *all* configurations as faults are usually caused by feature interactions among few features [Abal et al., 2018]. Thus, there are strategies that exploit this property of configuration options to analyze only a subset of configurations.

Sampling

Sampling (aka. *product-based* analyses) generate a subset of configurations with the goal of detecting faults by changing the configuration under test [Medeiros et al., 2016, Thüm et al., 2014, Varshosaz et al., 2018]. These approaches reuse existing analyses (e.g., testing or type checking) and directly apply them to the generated configurations. Sampling strategies can be divided into the three groups *Unsystematic*, *Systematic*, and *Complete* sampling as shown in Table 2.2. For an overview on sampling techniques and their classification, we refer to the work of Medeiros et al. [2016] and Varshosaz et al. [2018]

Unsystematic sampling is often used in practice due to the low additional testing effort and usually low number of tested configurations [Medeiros et al., 2015]. Developers usually use hand-picked configurations to test the system. Such configurations can only show that the functionality works for the specific configuration. However, they cannot detect unexpected feature interactions. Additionally, the system may be tested with few randomly generated configurations. *Random sampling* aims to cover the whole configuration space evenly [Al-Hajjaji et al., 2016b, Ensan et al., 2012, Henard et al., 2014, Liebig et al., 2013]. However, such a sampling approach is difficult in the presence of constraints. Thus, the generated configurations tend to test certain options more often than others.

Systematic sampling approaches aim to generates specific configurations to achieve specific coverage goals. A common approach is to test a single configuration with all or most options enabled (aka. *all-yes*). This configuration is used to cover most functionalities. However, as feature interaction faults often appear because some options are disabled, such a configuration will miss many feature interaction faults [Abal et al., 2018]. Another sampling approach called *combinatorial interaction testing* (aka. *t-wise testing*) generates configurations that contain *all* interactions among t options [Nie and Leung, 2011]. This approach guarantees to cover all interactions among t options (i.e., all combinations of possible selections among any combination of t options). As a single configuration can cover many interactions at the same time, the number of generated configuration is usually much lower than the number of possible configurations. Combinatorial interactions testing assumes that only few options interact with each other at a time (i.e. options are mostly orthogonal) [Abal et al., 2018, Cabral et al., 2010, Cohen et al., 2007, Garvin and Cohen, 2011, Kuhn et al., 2004, Medeiros et al., 2016, Nguyen et al., 2014b, Nhlabatsi et al., 2008, Nie and Leung, 2011]. The idea comes from the practical observation that most interaction faults are caused by interactions of three options and at most six options cause faults [Abal et al., 2018, Garvin and Cohen, 2011, Kuhn et al., 2004]. Combinatorial interaction testing usually only scales to cover low interaction degrees with t smaller than four. For higher t the number of generated configurations and the time to generate them is usually too high for practical use, especially if options have constrains among each other (e.g., due to a feature model) [Johansen et al., 2012, Kang et al., 1990]. There are approaches to improve the efficiency of sampling by cover interactions earlier (e.g., by reordering the configurations) and aborting the analyses based on time constraints [Al-Hajjaji et al., 2016a, 2017]. Such approaches, however, cannot guarantee to find all interactions.

Recent studies have shown that other sampling algorithms can be more efficient (i.e., they find more faults with fewer configurations) then combinatorial interaction testing [Abal et al., 2018, Medeiros et al., 2016]. Especially the approach *one-disabled* was able to detect most interaction bugs by only generating at most t configurations for t options. One disabled generates configurations such that one option is disabled while all or most other options are enabled (the number of enabled options may be restricted due to constraints). While the approach cannot guarantee that it will find all interactions are found it was shown that it can detect most interaction bugs reported in four large configurable systems [Abal et al., 2018].

Statement coverage takes the source code into account to generate configurations such that all code elements are included at least once [Tartler et al., 2012]. This approach is however designed for variability implemented conditional compilation using preprocessors and for static analysis. When dealing with runtime variability, usually all code is included, and code is enabled or disabled based on runtime decisions. Thus, it is unclear which parts of the code is affected by options.

Configuration complete sampling approaches, generate configurations that can detect all faults. Thus, a brute-force approach that tests all configurations covers the whole configuration space [Halin et al., 2017]. Complete sampling approaches detect the same faults as this brute-force approach, however without generating all configurations. When executing a test case some options may not have an effect on the execution or they depend on each other. This observation can be exploited to generate fewer configurations (i.e., if the option A is not affecting the execution then it is not necessary to test a configuration where A is selected and one where it is deselected) [Kim et al., 2011, 2013]. The approach of *SPLat* generates configurations by dynamically observing the usage of options and generates only configurations for which the execution differs [Kim et al., 2013]. Ideally, SPLat only needs to test few configurations if only few options are affected, and options depend on each other. However, if options are implemented orthogonal all configurations need to be tested. Thus, Souto et al. [2017] propose a hybrid approach (S-SPLat) that combines the ideas of splat with incomplete systematic approaches, such as *one-enabled*. S-SPLat solves the problem that black-box sampling approaches test for combinations of options and combinations thereof that are not relevant for a test case.

Sharing

Despite the high number of configurations there is hope for efficient *configuration complete* analysis. Such analyses are able to detect all feature interactions similar to a brute-force approach while dealing with the exponential number of configurations. The idea of sharing-based (aka. family-based, variability-aware, or variational) analyses is to reduce the analysis effort by sharing parts of the analyses among configurations. Sharing is achieved by the observation that similar configurations also behave similar.

There are two ways to achieve sharing: First, applying existing analysis that are already able to reason about multiple executions, such as model checking [Clarke et al., 1999, Meinicke, 2013, Thüm et al., 2014] and symbolic execution [Baldoni et al., 2018, Clarke, 1976, King, 1976, Reisner et al., 2010], and second, lifting an existing analysis

to explicitly handle variability to share redundancies (i.e., *variability-aware* or *variational*), such as for type checking [Apel et al., 2012, Kästner et al., 2012a], model checking [von Rhein et al., 2011], and dynamic analysis [Kim et al., 2012, Meinicke, 2014, Meinicke et al., 2016, Nguyen et al., 2014b, Wong et al., 2018b]. In Chapter 4, we show how interactions impact the efficiency of different approaches and how sharing enables to scale testing for exponential configuration spaces [Meinicke et al., 2016].

In Table 2.2, we summarize testing strategies for configurable systems. We further categorize the strategies by black-box (highlighted with •) and white-box approaches. Black-box approaches do not consider the implementation of the system under test. Thus, they may test configurations redundantly. In contrast, white-box approaches exploit the implementation of variability to reduce the number of configurations to test or enable sharing among them.

Sampling			Sharing
Unsystematic	Systematic	Complete	variational
•hand-picked	•t-wise	•brute-force	○variational execution
•random	•one-enabled	○SPLat	○shared execution
	•allyes		○JPF-BDD
	○coverage		
	○S-SPLat		

Table 2.2: Overview on testing strategies for configurable systems. Strategies marked with • are black-box approaches that do not consider the implementation of the system under test. Strategies marked with ○ are white-box approaches that considder the implementation and can thus be optimized to sample fewer configurations or share redundancies among them.

2.2 Running example

In Listing 2.1, we introduce our running example to illustrate challenges regarding testing, debugging and code comprehension, that arise due to feature interactions. We reuse the example called *GameScreen* from a study by Melo et al. [2016] which is based on BestLap [Ribeiro et al., 2014]. Each chapter refers to this example to illustrate our solutions.

The program has three configuration options *blue*, *red*, and *green* that can be selected independently. If the options blue and red are true, the executions will throw an exception (see Figure 2.1). In the original study [Melo et al., 2016], participants were asked to understand the program, to figure out whether the program has a fault and in which configurations the fault appears. Melo et al. [2016] reported an average bug finding time for this program of 10 minutes while it took only four minutes for the same program without variability. This suggests that most of the time is spent due to the increased complexity of only three options.

The example emphasizes that exception traces are usually not sufficient to understand a fault as they often do not contain the cause of the fault. The exception

```
1  public class GameScreen {
2    private static boolean blue, red, green;
3
4    private static final int PERFECT_CUREVE = 4;
5    private static final int PERFECT_STRAIGHT = 1;
6    private static final int TIME_BONUS = 2;
7
8    int totalScore = 0;
9    int penalty = 0;
10
11   public static void main(String[] args) {
12     GameScreen game = new GameScreen();
13     if (blue) {
14       game.setPenalty(10);
15     }
16     game.computeLevelScore();
17   }
18
19   private void setPenalty(int penalty) {
20     this.penalty = penalty;
21   }
22
23   private void computeLevelScore() {
24     assert totalScore == 0;
25     totalScore = PERFECT_CUREVE + PERFECT_STRAIGHT;
26     if (green) {
27       totalScore += TIME_BONUS;
28     }
29     if (blue) {
30       totalScore -= penalty;
31     }
32     if (blue) {
33       assert totalScore < 0;
34     }
35     if (red) {
36       setScore(totalScore);
37     }
38     if (blue) {
39       if (totalScore >= 0)
40         throw new RuntimeException();
41     }
42   }
43
44   private void setScore(int score) {
45     if (score >= 0) {
46       totalScore = score;
47     } else {
48       totalScore = 0;
49     }
50   }
51 }
```

Listing 2.1: Running example GameScreen based on an experiment by Melo et al. [2016]

```
Exception in thread "main" java.lang.RuntimeException
    at GameScreen.computeLevelScore(GameScreen.java:40)
    at GameScreen.main(GameScreen.java:16)
```

Figure 2.1: Exception thrown in GameScreen due to the interaction $red \wedge blue$.

is thrown because the value of `totalScore` was set to 0 in Line 48 of the method `setScore` only under condition *red*. As this method already returned it is not part of the exception trace. Debugging support exist to enrich the exception trace with

information of such method calls [Ohmann and Liblit, 2017]. This however, increases the amount of data a developer needs to deal with when debugging.

In summary, the example illustrates why configurable systems are hard to *test* as the faults may be only thrown for certain configurations, and why they are hard to *debug* as effects of options and their interactions are difficult to track.

In Chapter 3, we show how variational execution can efficiently detect such interaction faults due to its ability of aggressively sharing redundancies among executions [Meinicke, 2014]. In Chapter 4, we present our work where we explore characteristics of feature interactions and their implications on efficient analyses [Meinicke et al., 2016]. In Chapter 5, we show how we help debugging such feature interactions bugs by aligning the executions of all configurations, called variational traces.

2.3 Summary

In this section, we discussed challenges in configurable systems due to feature interactions. We discussed state of the art for detecting feature interactions, and why they are often not sufficient. We discussed the difficulties to code comprehension due to variability and how state of the art aims to approach these. We finally introduced our running example that we will use to illustrate our solutions to help with *testing* and *debugging* configurable systems.

3. Variational Execution

This chapter is based on and shares material with the author's Master's thesis "VarexJ: A Variability-Aware Interpreter for Java Applications" [Meinicke, 2014] the VaMoS'17 paper "A Choice of Variational Stacks: Exploring Variational Data Structures" [Meng et al., 2017], and the OOPSLA'18 paper "Faster Variational Execution with Transparent Bytecode Transformation" [Wong et al., 2018b].

In this chapter, we present our work on *variational execution*, which is not the main contribution of this thesis, but its foundation. Variational execution is a dynamic analysis that allows to efficiently execute all configurations of a program. It essentially executes all configurations at once while sharing common parts of the executions and data, which allows to observe variations caused by interactions.

Variational execution has similarities with *symbolic execution* [Clarke, 1976, King, 1976]: both approaches aim to execute the program for large input spaces. However, symbolic execution operates on symbolic values, such as α to represent all integers, which are often *infinite* and can lead to expensive constraint solving [Clarke, 1976, King, 1976]. Variational execution operates always on concrete values instead, and thus, does not require specialized computations [Meinicke, 2014, Nguyen et al., 2014b, Wong et al., 2018b]. Due to inexpensive computations and near optimal sharing of memory and instructions, variational execution can scale to explore large configuration spaces [Meinicke et al., 2016, Nguyen et al., 2014b, Wong et al., 2018b].

As memory is shared among all configurations and because all variations are stored in data, we can trace each variation of the execution back to the inputs that cause it. This opens opportunities for variational execution as dynamic analysis: we can observe effects and interactions among inputs directly. We use variational execution to characterize interactions [Meinicke et al., 2016] (Chapter 4) and to explain why certain inputs interact and cause faults [Meinicke et al., 2018] (Chapter 5).

The goal of this research is to aid researchers studying interactions among many executions by providing an efficient dynamic analysis – variational execution.

3.1 Choice Calculus

The *choice calculus* is a central concept in variational execution [Erwig and Walkingshaw, 2011a, 2013, Walkingshaw et al., 2014]. The choice calculus enables to store and compute with differences in data that depend on options. We refer to such data differences as *conditional values*. Variational execution uses conditional values to represent differences in data and variability contexts, (i.e., propositional formulas, such as $A \wedge B$) to compute with this data under restricted configuration spaces. These concepts allow variational execution to maximize sharing in data and execution.

3.1.1 Variational Data Types

A *conditional value* (aka. choice value) is a multi-value that represents a mapping of concrete values to corresponding configurations using variability contexts [Erwig and Walkingshaw, 2011a,b]. That is, variables, such as field and local variables, can take multiple different values at the same time depending on options. This way the value differences can be stored while preserving its context (i.e., the mapping to the corresponding configurations). Conditional values can be represented as different *variational data types*, such as tag trees, formula tree or formula maps [Walkingshaw et al., 2014].

We illustrate how the different data types work to show their advantages and disadvantages when storing conditional data using the example in Listing 3.1. The example is designed to emphasize challenges of efficiently storing and computing with redundancies in the variational data structures. In the example, we first increment the value of `X` by 1 in *STEP 1* under condition $A \wedge B$. In *STEP 2*, we then multiply this value by two under condition C.

```
1 X = 1
2 STEP 1: if (A & B) X = X + 1
3 STEP 2: if (C) X = X * 2
```

Listing 3.1: Example to illustrate variational data types.

Tag Tree

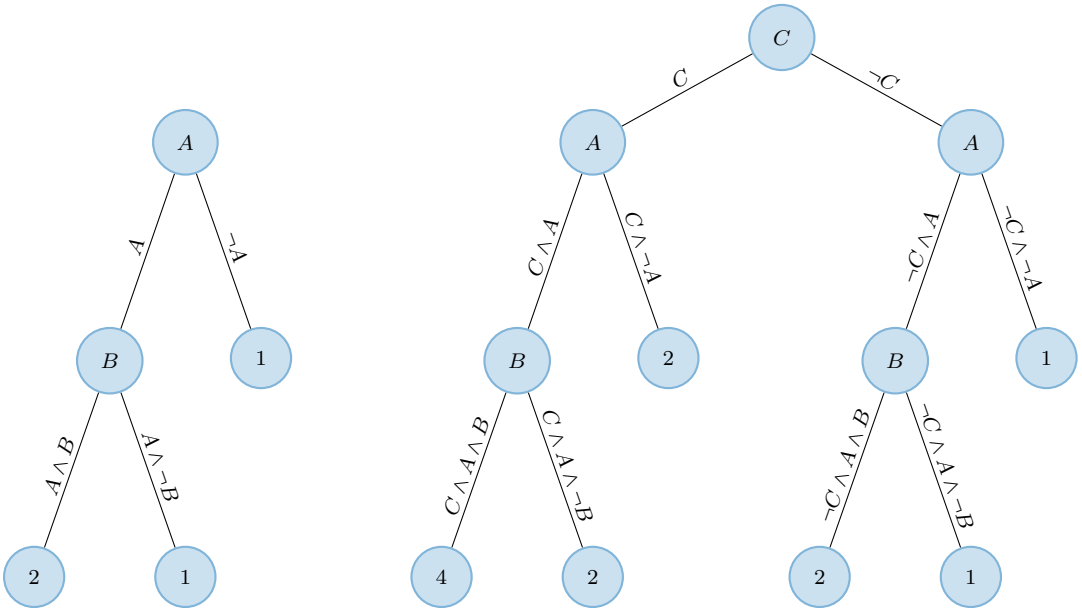
A tag tree is a tree structure in which the nodes split the configuration space depending on the selection of a single feature (tag). The leaves of the tree hold the values of the conditional value. A tag tree has the advantage that it does not require a SAT solver due to its simplicity. However, a tag tree might need to duplicate entries to represent the conditional value.

We illustrate tag trees in Figure 3.1. As shown, the tree splits the configuration space depending on single options. The context on the edges is implicit and for illustration purposes only. As shown, even to represent simple conditional values after STEP 1, tag trees need to duplicate values (i.e., the value 1 is contained twice). After STEP 2, the duplication increases even further: the value 2 is contained three times and 1 is contained twice. Note that the tree can be rearranged to represent the same conditional value. For example, instead of creating a C node on the top, we could

create C-nodes at each leaf. Reordering may influence the size of the tree and the number of duplications.

The duplication has effects on memory consumption. More severely however, computations with tag trees become expensive when the same operation needs to be applied multiple times redundantly. The advantage of tag trees comes if sat solving can be avoided. For example, if we want to apply an operation under condition A, we can take the true branch of A-nodes or create a new A-node at the end.

Austin and Flanagan [2012a] use tag trees to represent data differences in their work on dynamic information flow. As their applications usually only consider few options (e.g., high and low confidentiality), tag trees are an appropriate data structure as they contain only few redundancies but avoid satisfiability solving.



(a) STEP 1 (b) STEP 2

Figure 3.1: Tag Tree.

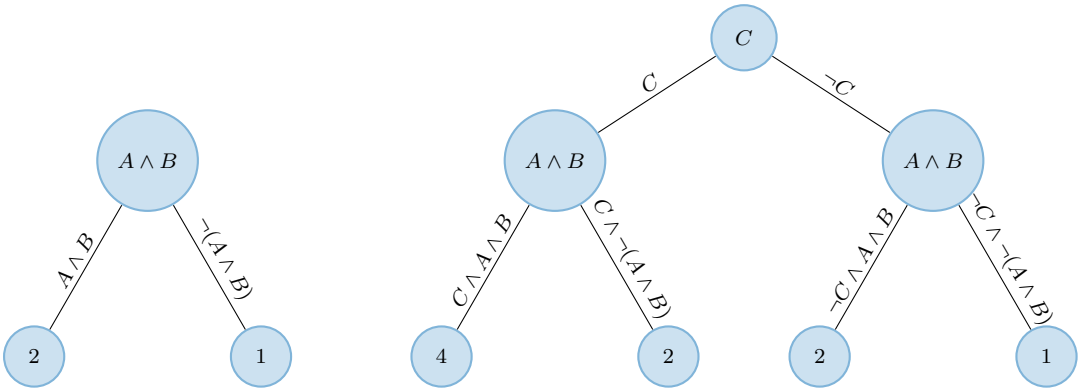
Formula Tree

Formula trees are another tree structure that use propositional formulas instead of single features in their nodes. This reduces the size of the tree as well as the number of duplications compared to tag trees. However, formula trees might still contain duplications. Reorganizing the tree structure can avoid all redundancies, which however can be costly as this requires comparing values to detect duplicates.

In Figure 3.2, we illustrate formula trees for the example of Listing 3.1. As shown, the tree can separate the configuration space depending on propositional formulas. The conditions on the edges are again implicit and for illustration purposes. Thus, for STEP 1, there is no duplication of values.

For STEP 2, we separate the value depending on option C and if C is true we multiply the value by 2. As shown, also formula trees may have duplications which are however far fewer than for tag tree. As the nodes can contain arbitrary propositional

formulas, a reorganization of the tree is possible to have each value only once. In our experience, only simple restructuring has positive effects the performance when computing with conditional values. If the computation that is performed on the values is expensive it is rather worth transforming the formula tree into a formula map so that the operation is not performed redundantly.



(a) STEP 1 (b) STEP 2

Figure 3.2: Formula Tree.

Formula Map

A formula map uses a direct mapping of concrete values to propositional formulas to represent conditional values. The formula map avoids all redundancies by design. Thus, computations with formula trees do not need to be performed redundantly. However, the costs of construction and manipulation is usually higher than for tree structures.

In Figure 3.3, we illustrate formula maps for the example. As shown in STEP 2 the formulas may get more complicated than for the tree structures which causes overhead for SAT solving. In our experience, we found that formula maps are efficient if they need to represent many different values, which often causes redundancies in formula trees. However, if they only represent few values, trees are often faster due to their simpler construction and manipulation [Meinicke, 2014].

2	$A \wedge B$
1	$\neg(A \wedge B)$

(a) STEP 1

4	$C \wedge A \wedge B$
2	$C \wedge \neg(A \wedge B) \vee (\neg C \wedge A \wedge B)$
1	$\neg C \wedge \neg(A \wedge B)$

(b) STEP 2

Figure 3.3: Formula Map.

A fine-grained storage of data differences in variational data structures enables main parts of our research on studying how options interact. We can observe how options interact to study feature interactions in highly configurable systems [Meinicke et al., 2016] (see Chapter 4), and we can explain how feature interaction cause unexpected behavior [Meinicke et al., 2018, Soares et al., 2018] (see Chapter 5).

3.1.2 Programming with Conditional Values

Variational programming allows to implement programs using conditional values [Erwig and Walkingshaw, 2011a, 2013]. This way we can explicitly calculates with variations. In variational execution, we use variational programming to represent data differences and to apply operations on them [Meinicke, 2014, Nguyen et al., 2014b, Wong et al., 2018b].

The advantage of conditional values over symbolic values is that the internal values are always concrete and that only the variability context is symbolic. Thus, in contrast to calculating with symbolic values, we can apply existing functions to all single values of the conditional value.

A simple way of calculating with conditional values is to apply a function to all values. In the choice calculus we use an operation *map* to apply a function $f(x)$ to all entries. The map function works as illustrated below:

$$V[T] \bullet f[T,R] \Rightarrow V[R] \tag{3.1}$$
$$Ch(A,a,b) \bullet f(x) \Rightarrow Ch(A,f(a),f(b)) \tag{3.2}$$
$$Ch(A,1,2) \bullet x*3 \Rightarrow Ch(A,3,6) \tag{3.3}$$

When calculating with conditional values it is more common to apply a function in specific contexts. To do so, we use the operation $flatMap$. $flatMap$ applies a function which takes a concrete value and returns a variational one. This way we can apply a function $f(x)$ under specific contexts only. The operation $flatMap$ works as illustrated below:

$$V[T] \bullet f[T,V[R]] \Rightarrow V[R] \tag{3.4}$$
$$One(a) \bullet f(x) \Rightarrow f(a) \tag{3.5}$$
$$Ch(A,a,b) \bullet f(x) \Rightarrow Ch(A,a \bullet f(x), b \bullet f(x))) \tag{3.6}$$
$$Ch(A,1,2) \bullet x \Rightarrow Ch(A,x*3,x) \Rightarrow Ch(A,Ch(A,3,1),Ch(A,3,6)) \tag{3.7}$$
$$\Rightarrow Choice(A,3,2) \tag{3.8}$$
$$Ch(A,1,2) \bullet x \Rightarrow Ch(B,x*3,x) \Rightarrow Ch(A,Ch(B,3,1),Ch(B,3,6)) \tag{3.9}$$

As shown, the functions $x*3$ are applied in restricted context A and B, respectively. When using the map functions we can see that we do not need to change the functions (e.g., $x*3$) as they always operates on concrete values.

3.2 Variational Execution

Variational execution aims to maximally share redundant executions using *conditional values* and *variability contexts* (i.e., a propositional formula defining under which condition an instruction is applied), at the cost of additional overhead for each computation [Kästner et al., 2012b, Meinicke, 2014, Nguyen et al., 2014b, Wong et al., 2018b]. Variational execution keeps track of all data using conditional values, which enables a fine-grained representation of shared data. If, for example, the value of a field differs among configurations, the values are stored as a choice in

the field, but other fields of the same object are shared for the entire configuration space. Instead of splitting the entire heap as usually for software model checkers, variations are stored locally. When computing with data, we only compute with distinct values of all inputs, of which there are typically much fewer than configurations in the configuration space. Furthermore, the compact representation using variability contexts in choices provides us with a way to track where options interact. This enables opportunities for variational execution as dynamic analysis. We can essentially monitor the program execution and compare the states among all configurations. We use this ability in the following chapters to measure interactions and to help understanding interactions [Meinicke et al., 2016, 2018].

In variational execution, options occur only in variability contexts of choices, but all values are concrete. In contrast to symbolic execution, symbolic configuration decisions do not intermix with concrete values. Hence, all computations are performed with concrete values. This separation of concrete values and symbols enables computations without the undecidability issues from abstractions in symbolic execution, therefore we rely on variational execution in our study.

Variational execution maximizes sharing of redundant calculations in two ways: First, variational execution achieves instruction-level sharing among control flows of all possible configurations. Second, the difference between program states is represented compactly using choices, such that small differences in local variables or heap objects can be represented without splitting the entire program state. In this way, variational execution achieves fine-grained sharing among all executions. To achieve variational execution, we need to handle the control flow to share executions and data storage to handle redundancies in data.

Variational Data

A main idea of variational execution is a fine-grained storage of variations in data using choice values. This means that all data of the program needs to be stored using choice values. This requires transforming local variables, parameters, fields, and values on the operand stack into choice values. This allows to store differences caused by options independent from each other which allows us to avoid the exponential explosion when representing conditional program states.

Variational Scheduling

In variational execution, the program counter of a method can point to different instructions at the same time under different contexts. For example, after a condition (e.g., an if-statement) that depends on a conditional value, variational execution jumps to two different instructions in the method, the next instruction and the instruction the condition refers to. Thus, the program counter points to different instructions based on corresponding contexts. The goal of variational scheduling is to execute the instructions in an optimal order (i.e., selecting the instruction of the conditional program counter that should be executed next) to maximize sharing while preserving correctness. In most cases, executing the instruction with the lowest index (i.e., the one that is furthest behind) is an optimal scheduling strategy. Wong et al. [2018b] have shown that such a variational scheduling is optimal in most cases.

Only few exceptions, such as complicated loops, may cause a non-optimal sharing depending on how the code is compiled. After selecting the next instruction, this instruction is executed under its corresponding context, such that the changes of the instruction are only applied in this context.

In addition to executing single instructions under certain contexts, also methods can have a context they belong to. Thus, similar to executing single instructions, executing a method has only effects to the program under the context the method is called in.

Some methods may not be executed variationally, such as native methods that cannot be lifted or interpreted directly. Thus, these methods cannot store variational data and they cannot be executed using variational scheduling. A practical solution is to execute these methods with all possible combinations of the choice parameters of the method. This is however, only possible if calling the method does not cause side effects (e.g., changing a global variable). If the method has side effects, executing it multiple times can cause incorrect behavior. In this case it is necessary to implement lifted versions of this method and its class, called model class. Model classes are common for modeling the environment in model checking and symbolic execution [d'Amorim et al., 2008, Sen et al., 2015, von Rhein et al., 2011] and other variants of variational execution [Yang et al., 2016]. These model classes are implemented such that they can handle conditional values.

In conclusion, variational execution combines two main concepts: variational data to achieve fine-grained sharing of data and variational scheduling to share the execution of instructions. In Section 3.3, we give a complete example illustrating variational execution based on our running example GameScreen.

3.3 Variational Execution of GameScreen

In Listing 3.2, we exemplify variational execution based on our running example (see Section 2.2) [Melo et al., 2016]. We annotated the example with arrows that show the conditional control flow, including the context under which each instruction is executed. Dotted arrows indicate the execution for the "not" context (e.g., not red). On the right side of the listing, we show the conditional value assignments, which are the values that are stored for variational execution. These values are either concrete values (e.g., 4 for `PERFECT_CURVE`) or choice values (e.g., $Choice(blue, true, false)$ for `blue`). Changes to variables are indicated in bold.

First, variational execution initializes all static fields. The fields `blue`, `red`, and `green` are annotated with `@Conditional`. We use this annotation to inform variational execution, that these fields should be treated as options Thus, we create features for each of the annotated fields and initialize them as `true` and `false` depending on the feature selection. For example, the field `blue` is initialized with $Choice(blue, true, false)$. Static field that are initialized with the same value for all configurations are initialized with a single concrete value.

When calling the main method, the execution is shared for all configurations (i.e., the execution context is True). At first, we create an object for GameScreen and initialize its non-static fields. Because the fields `totalScore` and `penalty` do not depend

```
1  public class GameScreen {
2    @Conditional private static boolean blue;                  -> Choice(blue, true, false)
3    @Conditional private static boolean red;                   -> Choice(red, true, false)
4    @Conditional private static boolean green;                 -> Choice(green, true, false)
5
6    private static final int PERFECT_CUREVE = 4;                                          -> 4
7    private static final int PERFECT_STRAIGHT = 1;                                        -> 1
8    private static final int TIME_BONUS = 2;                                              -> 2
9
10   int totalScore = 0;                                                                   -> 0
11   int penalty = 0;                                                                      -> 0
12
13   public static void main(String[] args) {
14     GameScreen game = new GameScreen();
15     if (blue) {                                          if blue
16       game.setPenalty(10);
17     }
18     game.computeLevelScore();
19   }
20
21   private void setPenalty(int penalty) {
22     this.penalty = penalty;                                  -> Choice(blue, 10,0)
23   }
24
25   private void computeLevelScore() {
26     assert totalScore == 0;
27     totalScore = PERFECT_CUREVE + PERFECT_STRAIGHT;                                     -> 5
28     if (green) {
29       totalScore += TIME_BONUS;                              -> Choice(green, 7,5)
30     }
31     if (blue) {
32       totalScore -= penalty;          -> Choice(blue, Choice(green, -3, -5),Choice(green, 7, 5))
33     }
34     if (blue) {
35       assert totalScore < 0;
36     }
37     if (red) {                                             if red
38       setScore(totalScore);
39     }
40     if (blue) {
41       if (totalScore >= 0)
42         throw new RuntimeException();
43     }
44     return;
45   }
46
47   private void setScore(int score) {
48     if (score >= 0) {
49       totalScore = score;              -> Choice(blue, Choice(green, -3, -5), Choice(green, 7, 5))
50     } else {
51       totalScore = 0;      -> Choice(blue, Choice(red, 0,Choice(green, -3, -5), Choice(green, 7, 5))
52     }
53   return;
54   }
55 }
```

Listing 3.2: Variational Execution for the runnning Example GameScreen. The arrow indicate under which condition the statements are executed. The values on the right side show changes to variables during variational execution.

on variability, they are initialized with simple concrete values. The next instruction is an if-statement (Line 15) that depends on a conditional value `blue`, which splits the execution depending on blue. This makes the program counter conditional (i.e., the execution is split), pointing to Line 16 under condition *blue* and Line 18 for $\neg$*blue*. As Line 16 is furthers behind (i.e., it is the PC with the lowest index of the

conditional PC), we execute Line 16 next under condition *blue*. Thus, `setPenalty` is invoked under condition *blue* and `penalty` is set to 10 for *blue* and penalty becomes a choice value. After `setPenalty` returns, all program pointers are at Line 18 and `computeLevelscore` is executed for again under the shared context $True$.

The same splitting and joining process happens for the method `computeLevelscore` as indicated by the arrows and the conditional assignments. In Line 38, the method `setScore` is called under condition *red* with the conditional value of the filed `totalScore` as parameter. The method `setScore` essentially sets the value of totalScore to 0 if the value of score is smaller than 0. As the changes are only applied under context *red* and as the value of `totalScore` are negative under condition *blue*, `totalScore` becomes 0 under condition $blue \wedge red$.

Finally, the Lines 40-42 check whether `totalScore` positive under condition *blue*. As `totalScore` is 0 under context $blue \wedge red$, a runtime exception is thrown for configurations with $blue \wedge red$. All other configurations do not fail and return in Line 44.

In Figure 3.4, we show the conditional exception thrown by variational exception [Meinicke, 2014]. As variational execution is aware of the condition under which the exception is thrown, the output is annotated with this additional information.

```
if (blue & red):
Exception in thread "main" java.lang.RuntimeException
    at GameScreen.computeLevelScore(GameScreen.java:42)
    at GameScreen.main(GameScreen.java:18)
```

Figure 3.4: Conditional console output for the exception in GameScreen due to the interaction $blue \wedge red$.

3.4 Implementations

In this section, we give a brief overview on current implementations of variational execution. There are two ways of implementing variational execution: a variational interpreter that executed the code directly, but assumes that certain inputs are conditional, and transformation of the program such that it behaves as if it would be executed variationally. We briefly introduce a tool for each type, namely VarexJ as a variational interpreter [Meinicke, 2014] and VarexC as variational bytecode transformation [Wong et al., 2018b].

VarexJ: Variational Interpreter

A *variational interpreter* is a tool that executes the program variationally. A variational interpreter usually extends an existing interpreter with variational scheduling and variational data [Kästner et al., 2012b, Meinicke, 2014, Nguyen et al., 2014b]. Previous variational interpreter have severe limitations: they are either only written for a toy language as proof of concept [Kästner et al., 2012b] or have limited language support [Nguyen et al., 2014a].

Our goal was to study variational execution for larger systems for a complete language. We chose Java as its specification is well defined. We implemented our variational interpreter VarexJ [Meinicke, 2014] on top of Java Pathfinder's [Havelund and Pressburger, 2000] interpreter for Java Bytecode. Even though Java Pathfinder is a software model checker, it is essentially a Java interpreter with model checking capabilities. To implement variational execution, we modified all bytecode instructions to compute with conditional data, we extended all shared data structures (e.g., the heap, the method frame) to store conditional data, and we implemented a specialized scheduling mechanism.

Those changes and the fact that VarexJ itself is written in Java creates a high runtime overhead for each instruction. As we build on top of Java Pathfinder, we inherit the same limitations, such as incomplete support for native methods and limited support for concurrency. This overhead and these limitations might forbid using VarexJ for practical use, but it is acceptable for our explorations. The advantage of extending an interpreter is that we can easily monitor each Java Bytecode instruction to observe interactions during runtime. To ensure the correctness of the implementation we used differential testing, we compared the executed instructions to the execution of all configurations for several of our subject systems [Kästner, 2017].

VarexC: Variational Bytecode Transformation

The goal of VarexC is to avoid the overhead and technical limitations of an interpreter by executing the program directly with a standard JVM [Wong et al., 2018b]. VarexC changes the bytecode such that when executed with a standard JVM, the program behaves as if it would have been executed with a variational interpreter. This means that VarexC transforms all data directly into Choice values (e.g., an int field becomes a Conditional<Integer>). To execute instructions conditionally, VarexC introduces additional instructions to the code so that map functions are called on conditional values. To achieve variational scheduling VarexC introduces additional jump instructions and structures the code into so called VBlocks. A VBlock is a set of instrucions that will always be executed under a shared context. To decide which block should be executed next, each VBlock has its own context and the block with the lowest index that has a satisfiable context will be executed next. Details on variational execution using bytecode transformation can be found in the OOPSLA paper [Wong et al., 2018b].

Note that we used VarexJ in all studies of this thesis [Meinicke et al., 2016, 2018], as VarexC was developed after the studies were performed. However, the findings and insights stay the same as the implementations are functionally equivalent.

3.5 Optimizations

The goal of this thesis it to study feature interaction in highly-configurable systems [Meinicke et al., 2016, 2018]. Thus, we worked on different ways to make variational execution applicable to larger systems [Lazarek, 2017, Meinicke et al., 2016, Meng et al., 2017, Wong et al., 2018b]. In this section, we discuss several ways to improve the performance that we explored during the development of VarexJ

and VarexC. These optimizations can help when implementing new engines for variational execution or similar techniques, such as symbolic executions or software model checking, in the future. In particular, efficient handling of variability is the main *challenge* of variational execution. Optimizations that handle variability more efficiently can have a high impact on performance. Handling variability can be improved in different ways: First, optimizing internal variational data structures used in the execution engine [Meng et al., 2017]. Second, using specialized variational data structures to represent data structures of the executed program [Lazarek, 2017, Walkingshaw et al., 2014]. Third, avoiding redundant satisfiability calls.

Internal Variational Data Structures

Variational data structures are designed to efficiently represent data for an exponential configuration space [Walkingshaw et al., 2014]. Walkingshaw et al. [2014] already outlined initial ideas for such data structures, on which our improvements are based on. We present our decorator-based approach to optimize variational data structures, exemplary on our implementation of the variational method frame [Meng et al., 2017].

The method frame is the central data structure of a JVM for computations. Thus, also in VarexJ a large amount of time is spend in the variational method frame. A method frame consists of two parts, the operand stack and local variables. As any data can be conditional, both the stack and the local variables need to be able to store conditional values. Efficient handling of these conditional values is crucial for the performance.

The main idea of our approach is that the efficiency of the implementation highly depends on the shape of the data it has to handle. Thus, beyond a general variational implementation for a stack frame that can handle arbitrary inputs, we aimed to provide optimized implementations for different shapes of variational data.

The optimizations are based on two observations on which data is processed when executing a program with variational execution. First, most of the time, the stack frame does not need to handle variability at all. That means that all calls to the stack frame are done in the same context and all the data are unconditional. Thus, an implementation that supports variability is unnecessary most of the time. To support such non-variational cases, we provide a Hybrid-Decorator that can switch between a lifted and an unlifted version of the stack frame.

The second observation is that bytecode instructions are connected. For example, values that are pushed under a condition are usually also popped again under the same condition. Thus, it is often sufficient to remember under which condition the current operands are pushed instead of representing all different stacks. Again, we support these cases with a Buffer-Decorator which adds a buffer that remembers the current context of the operands on top of a variational stack frame.

In VarexJ, we use both decorators at the same time as they are independent and provide performance improvements over a general variational stack frame. A systematic evaluation of variational stack frames can be found in our VaMoS paper [Meng et al., 2017].

External Variational Data Structures

Data structures used by the executed program that are modified by many different features can cause a main bottleneck for variational execution. In the worst case, such a shared data structure can cause an exponential explosion. For example, when adding a value to a list under two different independent contexts, 1 under context A and 2 under context B, variational execution has to represent all four different lists: $(\neg A \wedge \neg B : \{\emptyset\}, A : \{1\}, B : \{2\}, A \wedge B : \{1, 2\})$.

Lazarek [2017] explored how to efficiently represent such variational data structures in the program. Specialized variational representation of such data structures can handle changes under several different contexts. We extended VarexC with an implementations of a variational list including a specialized scheduling algorithm that allows to efficiently iterate on such lists [Lazarek, 2017, Wong et al., 2018b].

A further exploration and general solutions for variational data structures (e.g., an automated lifting algorithm) are still missing and open research. However, these data structures are necessary for variational execution to be efficiently applicable to arbitrary programs.

Avoiding redundant SAT calls

Another bottleneck of variational execution is the number of satisfiability calls and operations on propositional formulas. Each executed bytecode instruction requires multiple of these calls, to check the condition of the operation and for modifying data. Thus, even a perfectly optimized variational execution engine cannot become faster than the number of operations the library for propositional formulas can perform.

A main observation from our study on how feature interact in configurable systems [Meinicke et al., 2016] is that most of the operations on propositional formulas are performed redundantly. Thus, we can cache all operations, such as `and`, `not`, and `isSatisfiable`. Especially, `isSatisfiable` can be costly if the program uses a feature model. Caching can effectively reduce the effort for these operations to simple map calls. We implemented this optimization in both VarexJ, and VarexC. In VarexC we experienced a speedup of at least four times when caching SAT calls. Reusing previous constraint solutions is used to speed up symbolic execution engines, like our optimization [Baldoni et al., 2018, Cadar et al., 2008b].

In Figure 3.5, we compare the number of calls and distinct calls (i.e., the size of the cached calls) on propositional formulas. To estimate the effect of caching SAT calls, we used six commonly used programs from previous evaluations of variational execution [Meinicke, 2014, Meinicke et al., 2016, Wong et al., 2018b]. As shown in the Figure, the size of the cache is 1 to 6 orders of magnitude smaller than the number of calls. The highest effect can be observed in GPL due to mostly orthogonal features and long runs in few different contexts (i.e., few interactions).

In summary, variational execution has the potential to scale to large systems with large numbers of configurations by efficiently sharing redundancies among them. We explored several ways to increase sharing and improve performance. As variational execution is a new analysis technique, it has still potential for further improvements by future research, such as general solutions for variational data structures [Walkingshaw et al., 2014].

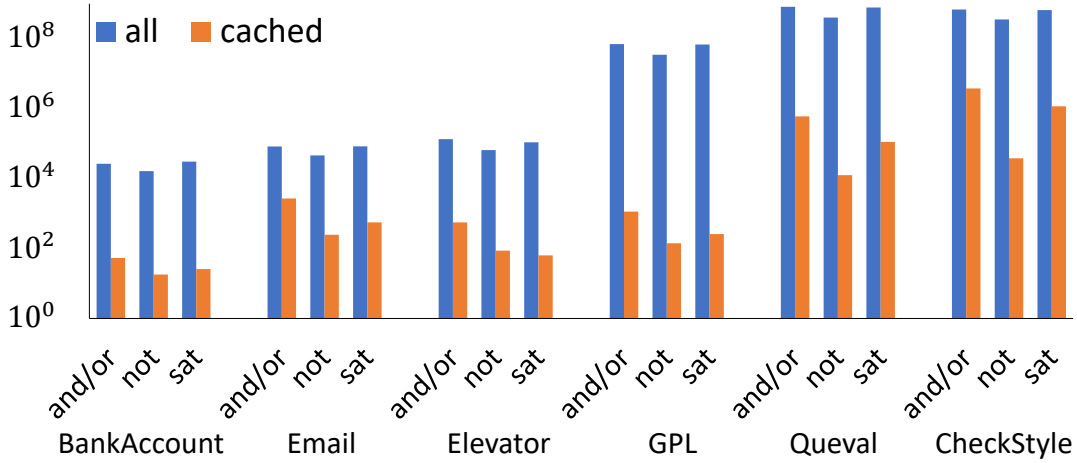


Figure 3.5: Number of SAT calls compared to number of distinct cached calls.

3.6 Related Work

Detecting faults caused by interactions of multiple options is challenging as it requires finding a configuration that reveals this fault. Recent research on highly-configurable systems has worked on solving the challenges that come with the configuration space explosion. As discussed in Chapter 2, there exist many approaches that test the system for a subset of configurations only [Abal et al., 2018, Cabral et al., 2010, Cohen et al., 2007, Garvin and Cohen, 2011, Kuhn et al., 2004, Medeiros et al., 2016, Nie and Leung, 2011]. In this section, we discuss work that is more closely related to variational execution in a sense that they share redundancies.

Initial work on variational execution was designed for a toy language as a proof of concept [Kästner et al., 2012b]. A second implementation is a variational interpreter for PHP that was built to analyze how 50 plugins interact in WordPress [Nguyen et al., 2014b]. However, due to the language design of PHP, the interpreter is incomplete and can hardly be used to analyze other systems. Our goal was to overcome these limitations and provide variational execution for a complete language to be able to study more diverse systems. Thus, we chose Java as it has a clear specification of instructions. With VarexJ [Meinicke, 2014, Meinicke et al., 2016] and VarexC [Wong et al., 2018b], we have two mature implementations with nearly complete language support, with only minor limitations.

There exist approaches with similarities to variational execution. Kim et al. [2012] propose shared-execution based on Java Pathfinder, which performs a form of variational execution, however not beyond method boundaries. The approach improves sharing, however as it requires to split the execution at method calls the ability to share are still very limited.

The approach called faceted execution is the similar idea as variational execution applied in the context of secure information flow analysis [Austin and Flanagan, 2012b, Austin et al., 2013, Yang et al., 2016]. Like variational execution, faceted execution uses faceted values (i.e., choice values) to represent data that differs across executions.

Coalescing execution [Sumner et al., 2011] uses vectors to represent multiple values, similar to choice values. However, the execution model scales only to few different inputs.

Multi-execution approaches run the program multiple times in parallel [De Groef et al., 2012, Devriese and Piessens, 2010, Hosek and Cadar, 2013, 2015, Kolbitsch et al., 2012, Maurer and Brumley, 2012, Su et al., 2007], however, without sharing redundancies among them. Thus, multi execution is often only used to analyze few executions (usually two).

Software model checking is designed to explore multiple program paths while shearing parts of the execution, by splitting and joining the executions. However, as executions can usually only be merged if the program states are equivalent, the states can rarely be merges, leading to the so called *state space explosion* [Ball et al., 2011, Beyer et al., 2007]. von Rhein et al. [2011] extended Java Pathfinder by abstracting configuration options as symbolic variables using BDDs. These options are then no longer part of the global state of the system. Thus, removing the options from the state enables to split the executions later and increase the probability of joining the executions again. However, as soon as options affect the global program state (e.g., a value of a field is changed depending on the selection of an option), JPF-BDD can no longer share the execution. There exist further model checking approaches that include features into the verification process [Asirelli et al., 2011, Classen et al., 2010, 2011, Lauenroth et al., 2009], which however only operate on models of systems and not on source code.

Variational execution shares ideas of dynamic symbolic execution [Clarke, 1976, King, 1976], but instead of calculating with symbolic values, variational execution always calculates with concrete values. Usually, symbolic execution has to split the execution and the state when branching (e.g., due to if-statements), while not being able to merge the execution again [Baldoni et al., 2018]. Symbolic execution engines that implement a fully symbolic memory, use if-then-else formulas (like choices [Erwig and Walkingshaw, 2011a]), and optimally merge branches potentially behave like variational execution [Baldoni et al., 2018, Sen et al., 2015]. Such symbolic execution engines reduce the effort of analyzing a system by sharing redundant computations, which, however, comes with the cost of more expensive constraint solving [Kuznetsov et al., 2012]. For example, the approach of MultiSE is closely related to variational execution [Sen et al., 2015]. MultiSE introduces a fine-grained representation of data differences, similar to choice values, which enables more sharing as it is no longer necessary to split the whole program state.

3.7 Conclusion

In this chapter, we presented variational-execution [Meinicke, 2014, Nguyen et al., 2014b, Wong et al., 2018b], which builds the foundation of this thesis. With variational execution we developed an efficient dynamic analysis that essentially aligns the executions of many configurations. We use this dynamic analysis throughout this thesis to (1) understand how options interact in highly-configurable systems (Chapter 4)[Meinicke et al., 2016] and (2) to help developers understanding interactions and their causes (see Chapter 5) [Meinicke et al., 2018]. We further presented

present recent our advancements of variational executions, such as optimizations for internal [Meng et al., 2017] and external variational data structures [Lazarek, 2017], and a new implementation that avoids the overhead of an interpreter-based approach [Wong et al., 2018b].

4. Measuring Interactions in Highly Configurable Systems

This chapter is based on the ASE'16 paper "On Essential Configuration Complexity: Measuring Interactions in Highly-Configurable Systems" [Meinicke et al., 2016].

Research in the area of configurable systems has developed many approaches to deal with the feature interaction problem, such as sampling. These approaches are built on certain assumptions about feature interactions, such as that only few feature interact, and that practically all faults can be detected by covering interactions among three features [Abal et al., 2018, Garvin and Cohen, 2011, Kuhn et al., 2004]. These assumptions come from indirect observations of how variability affects the system behavior (e.g., though bug reports) [Abal et al., 2018, Garvin and Cohen, 2011, Kuhn et al., 2004]. However, little is known on how features interact inside the system: *"Are all options orthogonal?"* or *"Do options interact randomly with each other?"* Such knowledge is helpful to develop better (e.g., configuration complete) analyses that scale to large configuration spaces as they complete and backup the assumptions on feature interactions that the analyses rely on.

Recent research started investigating how exactly options interact. Reisner et al. [2010] observed interaction through path constraints in symbolic execution. In their experiments they used symbolic execution to execute all configurations and to detect differences in the code coverage that depend on configuration options and interactions thereof. However, due to the scalability issues of symbolic execution, the experiments took 80 machine weeks to execute the 319 test cases of 10k LOC programs while exploring at most 30 configuration options.

Nguyen et al. [2014b] used variational execution to analyze WordPress, a blog post management system, and found that higher-order interactions exist that include up to 16 plugins. Their experiment was mainly a feasibility study that should show the scalability of variational execution. Only a small part of the study was dedicated analyzing interactions, such as the numbers of interacting options on data and control-flow.

Our goal is to directly measure how options interact during runtime to observe the behavior of feature interactions. We use variational execution which provides (nearly-)optimal sharing, similar to the study of Nguyen et al. [2014b]. The sharing potential of variational execution and the way how variational execution represents data and control-flow differences enables us to directly observe interactions. That is, we can observe how features interact in the control flow by observing under which condition instructions are executed, and we can observe interactions on data by analyzing how many options are involved in the choices that represent certain values. We further measure the *essential configuration complexity*: the configuration related differences in an execution that need to be explored given an optimal execution strategy.

Using these analyses, we developed three metrics that allows us to visualize how options interact. Specifically, the degree of the interactions on the control flow and how complex computations are depending on the number of alternatives that need to be explored (i.e., the essential configuration complexity). These metrics help us understanding how interactions affect different configuration complete analysis techniques that are proposed to analyze configurable systems, namely SPLat (sampling) [Kim et al., 2013], symbolic execution [Anand et al., 2007, Reisner et al., 2010], software model checking [Havelund and Pressburger, 2000, von Rhein et al., 2011], and variational execution [Meinicke, 2014, Nguyen et al., 2014b]. We developed five micro benchmarks that allow us to observe the effects of different interactions on tools that implement different sharing strategies. With these micro benchmarks we gain an understanding how different sharing strategies affect scalability.

We further apply our measurements to eight configurable systems to investigate the characteristics of interactions in actual programs. We analyzed the metrics and found, beyond others, that options are often orthogonal and interact locally, but high-degree interactions exist.

The goal of this research is to aid researchers gain a better understanding of how options interact during program execution by analyzing how data interacts in variational execution.

Overall, we contribute the following in this chapter:

- We develop three metrics that characterize essential configuration complexity, measuring how options interact within an execution.
- We design five benchmarks to study how state-of-the-art analysis approaches exploit interaction characteristics in exponential configuration spaces, exposing why certain approaches do not scale for certain kinds of interactions.
- We measure the configuration complexity for medium-sized systems, finding that the essential configuration complexity is low enough to enable configuration-complete analyses.
- We discuss common characteristics of interactions, providing more nuanced variants of current assumptions, which can, among others, encourage more efficient analyses of programs with large configuration spaces.

4.1 Measuring Feature Interactions

Sharing executions and compactly representing data differences, our dynamic analysis can directly collect data about interactions. Our execution over-approximates essential interaction complexity where sharing is suboptimal. Technically, we instrumented the execution of each Java bytecode instruction to collect data on interactions to measure three metrics: the control-flow interaction degree, the data interaction degree, and the interaction overhead. We exemplify the measurements for our running example GameScreen in 4.1a. Based on this execution, we generate metrics to on feature interactions shown in 4.1b, containing interactions overhead, interactions on data and interactions on the control flow.

With *control-flow interaction degree*, we measure configuration complexity on the control flow by assessing how many options need to be selected or deselected to execute the instruction at this point of the trace. The degree increases at control flow decisions that depend on a configuration option; in our example, the instruction in Line 42 is executed with context *blue*∧*red*, thus this instruction's control-flow interaction degree is two. As our analysis already tracks the variability context during execution, we merely need to log the number of options in the context for each executed instruction. In our plots, we visualize the control-flow interaction degree as a red line along the trace. A high value indicates part of an execution that is only contained in few configurations.

With *data interaction degree*, we measure configuration complexity on data by assessing on how many options the resulting value of an instruction depends. Considering variability, an instruction may need to be computed with alternative values and the result of the instruction may depend on one or multiple options, of which we report the number of distinct options affecting the value. For example, the expression computing `totalScore` in Line 51 results in five alternative values depending on three different options, resulting in a data interaction degree of three. We measure the degree by inspecting the result of every instruction during execution and plot it as a green bar along the trace. A high value indicates that some different results from a computation might be observable in few specific configurations only.

Finally, with *interaction overhead*, we measure the effort required to execute an instruction considering data variability in the instruction's inputs. If all inputs of an instruction have the same value in all configurations, we need to execute the instruction only once (baseline overhead 1). If one input has n alternative values in different configurations, we need to execute the instruction n times (overhead n). For instructions with multiple inputs (e.g., addition or method invocation), we need to consider all combinations of alternatives of all inputs (worst case overhead $n \times m$ for an instruction with two inputs with n and m alternatives respectively). In contrast to interaction degree measures, interaction overhead assesses the essential computational effort from alternative values, not how many options are involved. For example, in Line 32 the two values of `totalScore` are combined with the two values of `penalty` (overhead $four$). We compute the interaction overhead by inspecting the variability in all inputs of each instruction and plot it as blue bars along the trace. The interaction overhead is useful to compare essential complexity to the effort for executing a single configuration; comparing the aggregated overhead of all instructions with those of

a single execution allows us to assess how many additional instructions have been executed and how many instructions need to be repeated due to variability.

All three measures assess different aspects of configuration complexity. The interaction degree measures characterize interactions in control flow and data, whereas interaction overhead approximates the effort required for a configuration-complete analysis considering maximal sharing. The trace for our example in Figure 4.1 illustrates how the measures peak every time data from interactions is created or accessed.

4.2 Interaction Benchmarks

After introducing how we measure configuration complexity technically, we illustrate how certain kinds of interactions affect essential configuration complexity with a series of benchmarks. The benchmarks provide a sanity check for our measures of configuration complexity before we collect and interpret the measures on real-world systems. Additionally, they allow us to study how well existing sharing-based analysis tools exploit redundancies and which interaction characteristics they exploit. This enables us later to extrapolate which analysis tools can cope with characteristics found in real-world software systems.

We designed five benchmarks shown in Table 4.1 that each exhibit different interaction characteristics in a short execution. In the second column, we plot the measured configuration complexity. Additionally, we compare execution time, executed instructions, and memory consumption of five state-of-the-art analysis tools: SPLat [Kim et al., 2013], JPF-core [Havelund and Pressburger, 2000], JPF-BDD [von Rhein et al., 2011], JPF-symbolic [Anand et al., 2007], and VarexJ. We do not evaluate sampling-based strategies, as our benchmarks are specifically designed to produce high-degree interactions. Specifically, we address the following research question: **RQ 1: What are the effects of different kinds of interactions on the scalability and performance of state-of-the-art execution mechanisms?**

4.2.1 Experimental Setup

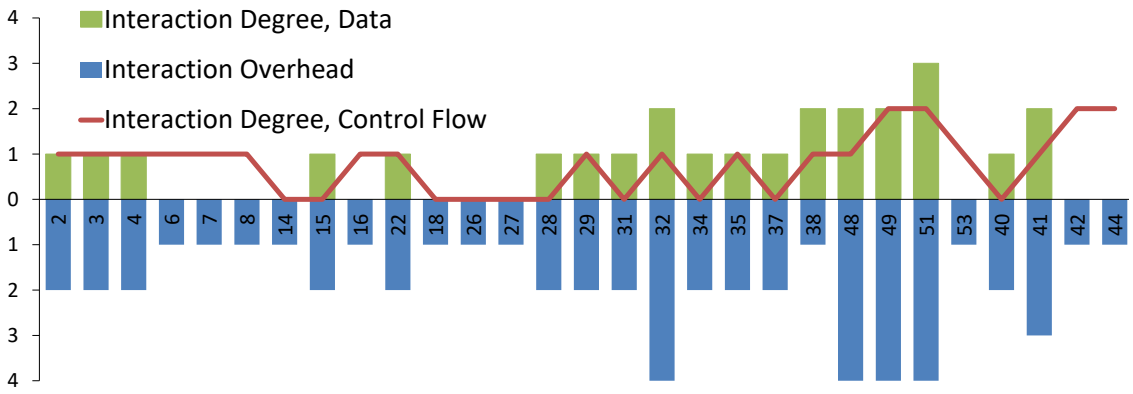
Evaluated Analysis Tools

We compare five state-of-the-art analysis tools that have been designed to efficiently execute a program over configuration spaces by fighting surface complexity through different kinds of sharing. Some of these tools have been designed originally for different purposes, such as model checking safety properties [Anand et al., 2007], but they have been suggested also for analyzing interactions or testing highly-configurable systems. We selected tools that represent different analysis and sharing strategies: identifying unnecessary options, software model checking, and symbolic execution. In addition, we use the uninstrumented version of our variational interpreter VarexJ a representative for variational execution. The tools are comparable in the sense that they all target Java and are mostly based on the same infrastructure, namely Java Pathfinder [Havelund and Pressburger, 2000].

Java Pathfinder (JPF-core) is a software model checker for Java Bytecode that handles bytecode instructions as transitions between states [Havelund and Pressburger,

```
1  public class GameScreen {
2    @Conditional private static boolean blue;                    -> Choice(blue, true, false)
3    @Conditional private static boolean red;                     -> Choice(red, true, false)
4    @Conditional private static boolean green;                   -> Choice(green, true, false)
5
6    private static final int PERFECT_CUREVE = 4;                                          -> 4
7    private static final int PERFECT_STRAIGHT = 1;                                        -> 1
8    private static final int TIME_BONUS = 2;                                              -> 2
9
10   int totalScore = 0;                                                                   -> 0
11   int penalty = 0;                                                                      -> 0
12
13   public static void main(String[] args) {
14     GameScreen game = new GameScreen();
15     if (blue) {                                  if blue
16       game.setPenalty(10);
17     }
18     game.computeLevelScore();
19   }
20
21   private void setPenalty(int penalty) {
22     this.penalty = penalty;                                          -> Choice(blue, 10,0)
23   }
24
25   private void computeLevelScore() {
26     assert totalScore == 0;
27     totalScore = PERFECT_CUREVE + PERFECT_STRAIGHT;                                     -> 5
28     if (green) {
29       totalScore += TIME_BONUS;                                      -> Choice(green, 7,5)
30     }
31     if (blue) {
32       totalScore -= penalty;          -> Choice(blue, Choice(green, -3, -5),Choice(green, 7, 5))
33     }
34     if (blue) {
35       assert totalScore < 0;
36     }
37     if (red) {                                   if red
38       setScore(totalScore);
39     }
40     if (blue) {
41       if (totalScore >= 0)
42         throw new RuntimeException();
43     }
44     return;
45   }
46
47   private void setScore(int score) {
48     if (score >= 0) {
49       totalScore = score;                -> Choice(blue, Choice(green, −3, −5), Choice(green, 7, 5))
50     } else {
51       totalScore = 0;          -> Choice(blue, Choice(red, 0,Choice(green, −3, −5), Choice(green, 7, 5))
52     }
53     return;
54   }
55 }
```

(a) Variational execution of GameScreen.



(b) Traces and interaction overhead for GameScreen.

Figure 4.1: Illustration of interaction metrics for GameScreen.

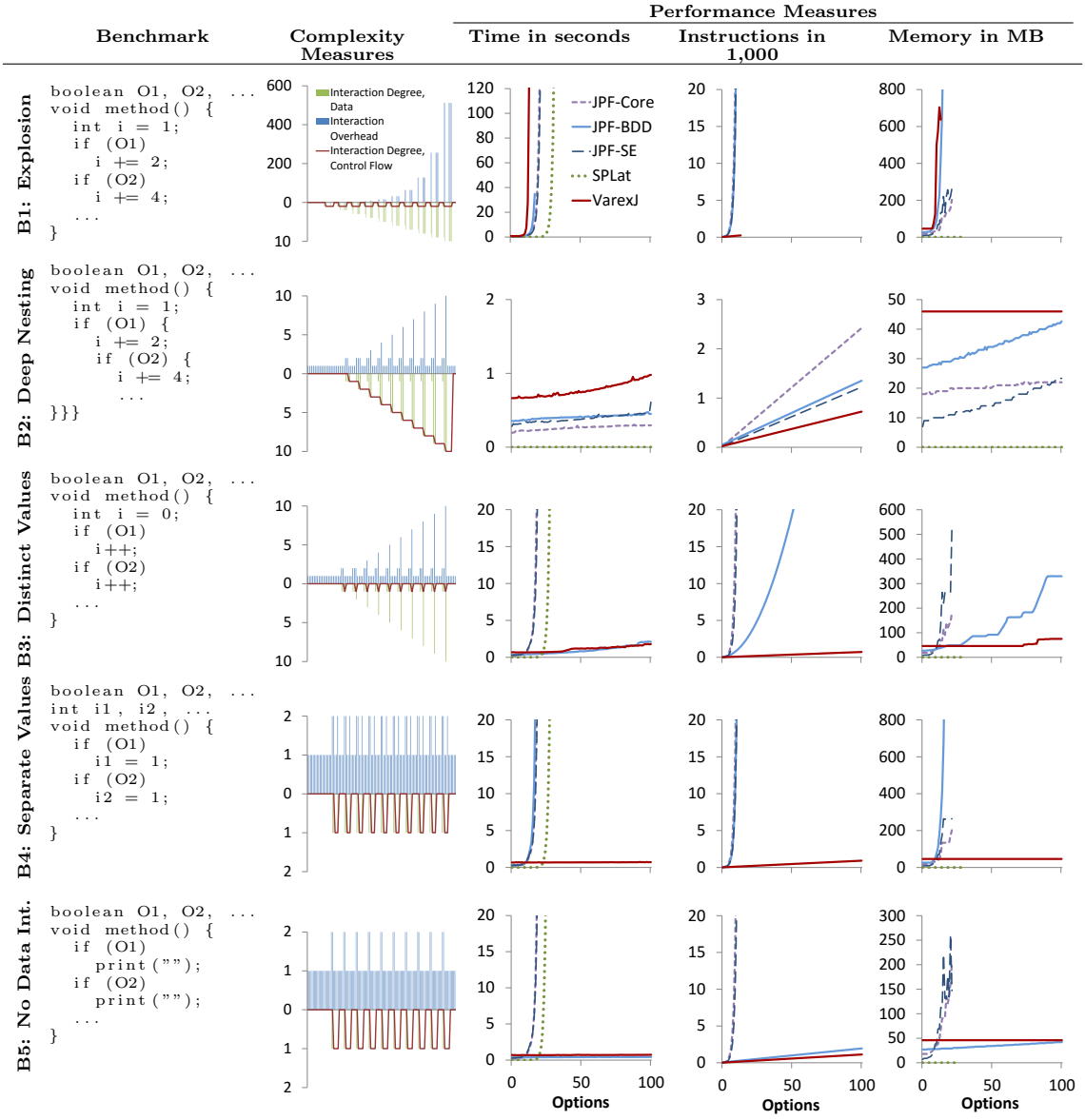


Table 4.1: Benchmarks to simulate different kinds of interactions (left). The diagrams in the second column illustrate interactions for each program measured using variaitonal execution. The three diagrams on the right show the performance results for time, executed instructions and memory consumptions for five analysis tools.

2000]. JPF-core can be used to split execution paths for boolean options and explore all possible paths. If all values of fields and variables are equivalent, JPF-core can join separated paths and share subsequent executions.

JPF-BDD extends JPF-core by separating tracking of boolean options [von Rhein et al., 2011]. By taking options out of the state, states can be merged if they differ only by options, increasing the chance for joining, and thus sharing.

JPF-symbolic is a symbolic extension of JPF-core [Anand et al., 2007], designed for test generation. If the execution splits (e.g., due to if-statements), the state is forked, but due to challenges in matching symbolic states, states are never merged. Other symbolic engines may yield better sharing by the memory fully symbolically,

which however may also result in more expansive computations [Baldoni et al., 2018, Sen et al., 2015].

Finally, SPLat instruments a program to dynamically detect which configuration options are used in an execution [Kim et al., 2013]. It reexecutes the program until all combinations of used options are explored. Although SPLat does not share any actual executions, it can narrow down the configuration space if only a subset of configurations has an effect on the execution trace (e.g., for unit tests). As the tool is not publicly available, we reimplemented it for Java.

Benchmarks and Metrics

We design five small benchmark programs characterizing favorable and critical cases for interactions among configuration options. We show all benchmarks in Table 4.1 and explain them and their rationale together with the results. All benchmarks are reduced to distill the interaction effect in a very concise setting. Each benchmark can be scaled in the number of involved configuration options, such that we can observe scalability with regard to the exponentially growing surface configuration complexity. We plot the *complexity measures* for an execution with 10 options to illustrate the general trend.

For each tool, we report the *performance measures* time, instructions, and memory consumptions for executing the benchmark with different numbers of options (0 to 100). We measured them all using internal metrics of Java Pathfinder and built a separate harness for SPLat. As we face an exponential problem, we terminate executions that exceed two minutes. To reduce measurement bias, we report the average of three runs.

4.2.2 Sharing Potential

For each benchmark, we discuss the interaction characteristic it simulates, reasons for the configuration complexity, and the performance measures indicating which tools scale.

Benchmark B1 (Explosion): We start with the worst case of interactions in which all options interact on the same value and yield a different result in every configuration. In such case, every exhaustive technique needs to track an exponential number of alternative values. As visible from the complexity measures, early and some later instructions (e.g., if statements) in the benchmark are affected by fewer configurations and can be shared. However, no tool can be expected to scale as they all face essential configuration complexity growing exponentially with the number of options as visible in all performance measures. If these kinds of interactions are common in practice, there would be little hope for configuration-complete analyses.

Benchmark B2 (Deep Nesting): Next, we explore the effect of dependencies among options, leading to a lower essential configuration complexity with a linear number of distinct execution traces. Whereas B1 had independent decisions for each option, resulting in 2^n execution traces, B2 models nested decisions, resulting in $n + 1$ execution traces for n options. The complexity measure shows the linear increase in overhead as additions are performed on values with increasingly many

alternative values. Also the interaction degree measures grow linear as more and more options need to be selected. Again, we can see that several instructions that do not manipulate variable i could be shared. Our performance measures show that this kind of interaction is well supported by all approaches. As all tools only split lazily where necessary, there are nearly linear increases with more options in all performance measures. Simpler tools outperform tools with higher constant overhead, as exploiting additional sharing has only marginal effects.

Benchmark B3 (Distinct Values): Sharing becomes feasible if interactions on a variable produce a small number of distinct values. In benchmark B3, each option increases a value by 1, resulting in $n + 1$ distinct values for n options. Therefore, essential configuration complexity grows linearly with the number of configurations. Our performance measurements indicate that JPF-core and JPF-symbolic require exponential effort as they need to split the execution on every *if* statement but cannot join them again; JPF-symbolic never joins and JPF-core cannot join as the values representing the options have different values in different states. Without data sharing, also SPLat requires exponential effort because all options have an independent effect on the execution trace. JPF-BDD and VarexJ both track the $n+1$ distinct values separately from the variations in configuration options, which enables them to perform closer to the linear growing essential configuration complexity. VarexJ executes fewer instructions and requires less memory than JPF-BDD by exploiting additional sharing, which however has no benefits for the execution time due to the additional overhead.

Benchmark B4 (Separate Values): If options affect disjoint parts of the state, essential configuration complexity can be very low. Benchmark B4 exhibits an interaction in which each option affects a different variable, without any data interaction. Despite an exponential number of execution traces and distinct states, each variable has only two alternative values (0 and 1) and, as such, the essential configuration complexity is low. As the performance measures show, JPF-core, JPF-symbolic, JPF-BDD, and SPLat all require exponential effort, as they do not exploit sharing for this interaction characteristic. All approaches split on each if-statement and none can join the states again. Even JPF-BDD cannot join, as non-option values differ across configurations. Only VarexJ approaches the low essential complexity.

Benchmark B5 (No Interactions on Data): Finally, we eliminate all data interactions, such that only control-flow interactions remain (i.e., an exponential number of different execution traces, all with the same states). Essential configuration complexity is low as in B4. JPF-BDD and VarexJ both execute each instruction on a single state without interaction overhead, as all variability of options is handled separately. In contrast, SPLat still needs to explore all execution traces and JPF-core and JPF-symbolic track different configuration values as part of their split state, resulting in exponential behavior.

Lessons Learned

Even when essential configuration complexity is low, missing to exploit suitable forms of sharing for certain characteristics of interactions can result in exponential execution efforts. A program with negligible essential complexity (e.g., without

System	LOC	Options	Config.	Instr. VA	$\sum$IO	μInstr.	Coverage
Jetty 7	145,421	7	128	12M	12M	12M	16%
Checkstyle	14,950	141	$>2^{135}$	407M	421M	198M	37%
Prevayler	8,975	8	256	28M	29M	15M	7%
QuEval	3,109	20	680	81M	94M	1M	45%
Elevator	730	6	20	89k	100k	29k	81%
GPL	662	15	146	17M	17M	9M	86%
Email	644	9	40	48k	55k	16k	96%
Mine Pump	296	6	64	14k	16k	14k	84%

Table 4.2: Subject systems analyzed for configuration complexity and their sizes in lines of code, number of options and configurations; number of instructions executed with VarexJ, the aggregated interaction overhead ($\sum$IO), the average number of instructions for single configurations (μInst.), and lower bound for line coverage reached with the sample method.

any data interaction, as in B4 (*Separate Values*)) can cause exponential behavior in state-of-the-art approaches. Finding such kind of interaction characteristics in real-world programs would be a great opportunity for quality assurance, as it indicates a high potential for configuration-complete analysis with sharing-based approaches.

4.3 Measuring Feature Interactions in Highly Configurable Systems

To assess essential configuration complexity of executions in real-world software, we applied variational execution to eight configurable systems shown in Table 4.2. We selected four configurable medium-sized systems from different domains, the http server Jetty 7, the in-memory database Prevayler, the static analysis tool Checkstyle, and the academic evaluation framework for database index structures QuEval [Schäler et al., 2013]. In addition, we included systems previously used as benchmarks in research on configurable systems: The systems MinePump [Kramer et al., 1983], E-Mail [Hall, 2005], and Elevator [Plath and Ryan, 2001] are small academic Java programs that were designed with many interacting options; GPL [Lopez-Herrejon and Batory, 2001] is a small-scale configurable graph library often used for evaluations in the product-line community. All these systems are executable with VarexJ.

To investigate interactions in configurable systems, we pose the following research question: **RQ 2: What is the essential configuration complexity of real-world software?** Particularly, we are interested in whether our measures for configuration complexity confirm current assumptions based on error reports and program outputs [Abal et al., 2018, Garvin and Cohen, 2011, Kuhn et al., 2004] or whether they provide additional insights.

Experimental Setup

We execute all subject systems over all configurations with VarexJ. For each system, we measure configuration complexity for a fixed standard input: a sample input

distributed with QuEval, a source file with 474 lines for Checkstyle, and a sample application provided with Prevayler. For Jetty, we deploy a web application that is capable of serving static content as well as running simple servlets. As the traces often contain several million instructions, we aggregate subsequent instructions in our plots. We share the evaluation setup together with our implementation.

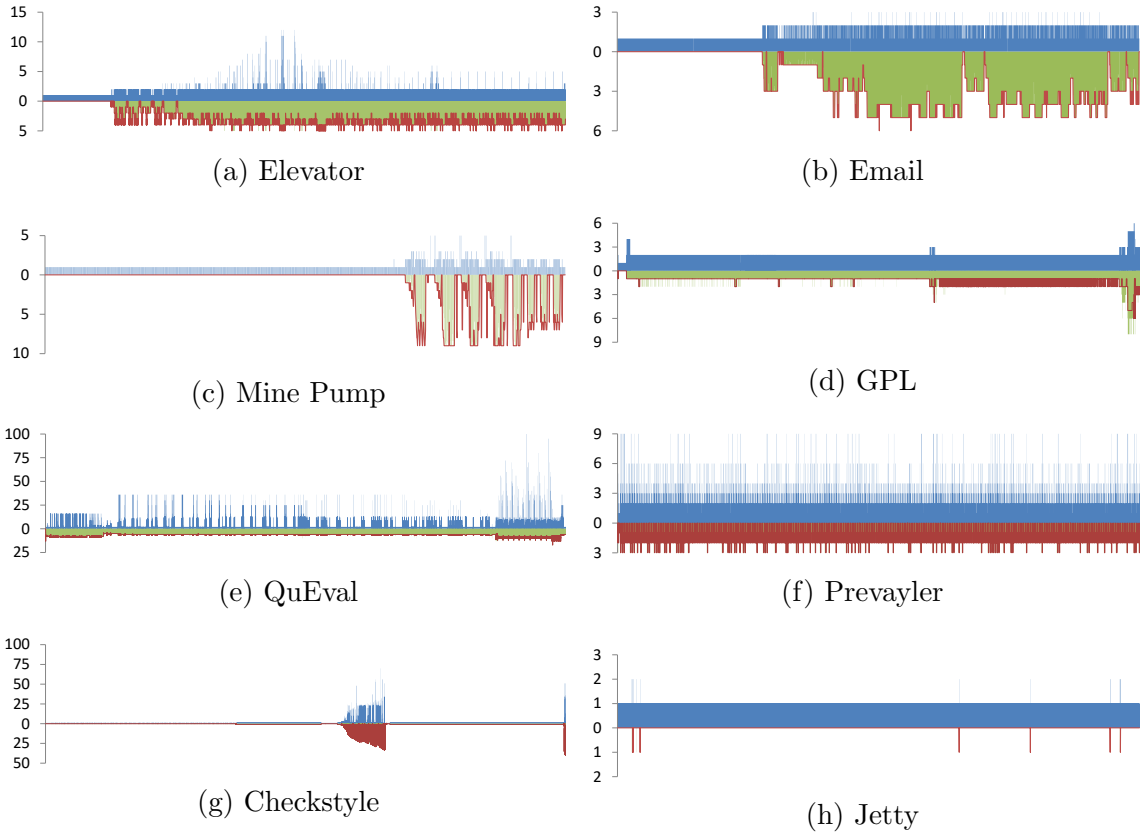


Figure 4.2: Traces and interaction overhead of variability-aware execution for larger software. Each bar represents the highest value per 1,000 instructions (per 10 for Elevator, Mine Pump and Email). Blue bars on top of the axis denote interaction overhead, green bars below the axis denote interactions degrees on data and the red line denotes the interaction degrees on the control flow.

Interactions in Real-World Software

We show the traces for the eight subject systems in Figure 4.2. In all systems, we can observe a small average interaction overhead throughout most of the trace and usually small interaction degrees (i.e., most instructions can be shared in large configuration spaces). The traces also show that options do not interact increasingly across the entire executions. Some individual results are noteworthy:

First, the Elevator system was specifically designed to exhibit many interactions [Plath and Ryan, 2001]. Its trace shows that several interactions on data cause an interaction overhead of up to 12. However, most instructions in the trace have an overhead of at most two. Many instructions are executed in restricted contexts though, requiring up to five options.

Second, GPL is a common system for evaluations in the product-line community, including prior studies of sharing and verification [Apel et al., 2013c, Kim et al., 2012]. The system has only some minor interactions with an interaction overhead of mostly two and a interaction degrees of mostly one option. Options do not interact at all for most parts of the trace. However, at the end of the execution up to eight options interact on the same data.

Third, we observed the strongest data interactions in QuEval. QuEval implements several database index structures which can be customized with several options, significantly changing the behavior of the entire system. The trace shows that there are long sequences with similar overhead in the execution. This is caused by separate processing of each index structure. Some values interact strongly causing an overhead of 100 (among 680 configurations). However, the trace still shows that high interaction degrees are rare, and many instructions can be shared after and between them. In QuEval, there are multiple interactions that cause high interaction degrees on data and control flow. Especially, in the last part of the execution, data interactions like benchmark B1 (*Explosion*) can be observed for a subset of the options.

Fourth, Checkstyle is a good example for a trace with particularly few interactions. The system implements many optional and independent checks that are not supposed to interact. However, the trace shows that there are still high degree interactions in Checkstyle, mostly caused by optional caching, resulting in a similar behavior as in the benchmark B3 (*Distinct Values*) in a subset of the trace. Also, in Jetty, we similarly observe that most options have only minimal influences on the trace; we found no interactions on data at all (possibly due to the simple test scenario we used).

Throughout all systems, we observe essential configuration complexity that is far lower than surface complexity may indicate. The amount of essential configuration complexity differs by system though from almost negligible (Checkstyle, Jetty, GPL), to medium (Elevator, Prevayler), to significant (QuEval). Comparing the aggregated interaction overhead with the average number of instructions executed without variability shown in Table 4.2 (see Section 4.1), we can see that a system executing close to essential configuration complexity would usually only have to execute $1 - 4$ times more instructions than an average execution of a single configuration. Only QuEval had a significant interaction overhead compared to an average execution of individual configurations, but that can be explained largely by executions for alternative options. In general, the overhead is much lower than the overhead of factor 20 to 2^{135} a brute-force approach would require and could potentially even beat some sampling strategies that reexecute each sampled configuration.

In its current form, due to the high overhead per instruction, VarexJ cannot achieve this speedup compared to a standard JVM.[1] However, our results indicate that essential complexity is low and there is hope for the community to develop efficient configuration-complete analysis techniques.

[1]When compared to executing a single configuration with VarexJ's own interpreter, we observe performance overheads between 1.0x (Jetty) and 9.7x (GPL) for most systems and 190x for QuEval, in line with the measured interaction overhead. Detailed performance measurements are outside the scope of this work, but can be found on our website.

Threats to Validity

Concrete results from our measurements should be generalized only carefully; our focus is on establishing metrics for configuration complexity, not on proving characteristics of programs in general practice. External validity is limited by the number and size of our subject systems. As described, we selected the small programs representing critical and paradigmatic cases, whereas we used convenience sampling for the medium-sized systems, primarily due to current technical limitations of our interpreter and the high engineering effort to execute further and larger systems. Our subject systems are diverse, but their characteristics may not generalize for other systems.

As described, we executed each system with only one input. Thus, we potentially miss interactions that occur only with other inputs. Nonetheless, we execute each program's main method with a representative input, which in each system covers all configuration options and a large amount of its code as the measured line coverage in Table 4.2 indicates.

To interpret our results, it is important to remember, as discussed in Chapter 2, that we define interactions as any differences during the execution triggered by options, not just externally visible differences or defects. This decision is deliberate to study interactions and execution methods in general, independent of defects they may cause [Abal et al., 2018, Garvin and Cohen, 2011, Kuhn et al., 2004].

4.4 Discussion: Characteristics of Interactions

In Section 4.2, we have shown that despite exponential surface complexity many kinds of interactions actually have low essential complexity, which can be exploited by suitable sharing-based analyses. In Section 4.3, we have subsequently shown that also real-world systems typically have a much lower essential complexity than it may appear on the surface. However, we have also seen that interactions in real-world systems have characteristics that are more nuanced than expected by existing approaches. Therefore, we conclude with a discussion of observed characteristics that may inform the design of future analysis approaches and may also be informative for developers concerned about interactions in their code.

We identify three main characteristics that are exploited (though not always explicitly) by existing analyses: irrelevant variability, orthogonal variability, and local variability.

Irrelevant variability: *Some options may not have any effect on an execution at all.* Even when a program has a large configuration space, some executions, such as test cases, may not even read certain configuration options. If no configuration of an execution ever reads a configuration option, we call such execution **unaffected** by the option. In addition, some options may never be read unless another option is (de)activated, in which the first option **depends** on the second. In both cases, the number of distinct executions is smaller than the exponential surface complexity indicates.

All sharing-based approaches exploit irrelevant variability, as shown with benchmark B2 (*Deep Nesting*). Although, irrelevant variability was attributed with significant

speedups for test cases in prior work [Kim et al., 2011, 2013], none of our real-world executions benefited from *unaffected* variability without rewriting the system to initialize options lazily (all options were always read and initialized). *Dependencies* reduced the search space, but never close to essential configuration complexity.

Orthogonal Variability: *Many options may not interact with each other.* Although potentially every option could interact with every other option, resulting in exponential surface complexity, a common assumption is that most options do not interact. We say two options are orthogonal if combining both options does not yield any new behavior that could not be explained by either option alone. Some options may be **strictly orthogonal** and not interact with any other option, but it is more common to assume **low interaction degrees** where each change can be explained by the interaction of at most two or three options.

The effectiveness of sampling strategies typically hinges on low interaction degrees, whereas most existing sharing-based approaches are rather inefficient in exploiting orthogonality, especially when options affect data, as apparent from benchmarks B4 (*Separate Values*) and B5 (*No Interactions on Data*). Our real-world executions confirm that many options are orthogonal, but also show that one should not rely on low interaction degrees alone: We found high interaction degrees (e.g., 40 in Checkstyle) in most systems, but also found that those involve some options while others remain mostly orthogonal. We argue that **rare high interaction degrees** is a more accurate characterization of interactions in real-world systems, encouraging research into configuration-complete analyses.

Local variability: *An option may affect control flow and data during an execution, but its effects might not spread across the entire execution trace, resulting in much lower essential configuration complexity than surface complexity.* With locality, we might need to invest more effort to execute part of the trace repeatedly for different configurations, but we can share effort in other parts.

Many sharing-based approaches exploit locality by sharing executions before the option's effect, and possibly also after (see Section 4.2). Existing sharing-based approaches differ in what forms of locality can be exploited though. Many approaches can share a common prefix of the execution trace (**prefix sharing**) and split late on the first instruction depending on an option. Some approaches can join after local instructions, if those instructions do not affect the state as in benchmark B5 (*No Interactions on Data)* (**strictly local**). Interactions that affect some state, that is, however, not read again subsequently (see benchmark B4 (*Separate Values*)) are rarely supported.

A much more common pattern in the observed real-world executions is what we call **scattered local**: Options affect the trace locally and cause some changes to the program's state, but many subsequent instructions can be shared before that changed state is accessed again. This is an effect, which we observed as gaps between peaks in the measures of our benchmarks and the real-world executions. In all cases, we see strong evidence of locality in that essential configuration complexity always returns to lower values after peaks.

	Unaffected	Depending	Strictly Orthogonal	Low Interaction D.	High Interaction D.	Prefix Sharing	Strictly Local	Scattered Local	Eng. & Runtime Ov.
Combinatorial testing			●	●					very low
SPLat	●	●							low
JPF	●	●	◐	◐	◐	●	◐		high
JPF-SE	●	●	◐	◐	◐	●			very high
JPF-BDD	●	●	◐	◐	◐	●	●	◐	high
VarexJ	●		●	●	●	●	●	●	very high
VarexC	●		●	●	●	●	●	●	high

Table 4.3: Interaction characteristics exploited by different analysis approaches. ●: exploited, ◐: partially exploited.

Outlook

We observed that essential configuration complexity is often low and exploiting irrelevant variability, orthogonal variability, and local variability is a promising avenue to scale analysis approaches. However, we also found that supporting the more nuanced characteristics of *rare high interaction degrees* and *scattered local* effects are essential for scaling sharing-based approaches to large configuration spaces.

We summarize which properties are supported by each of the discussed tools in Table 4.3. Currently, the tools that exploit more characteristics are also based on a more heavy-weight infrastructure (i.e., higher engineering effort for the analysis and higher runtime effort to execute individual instructions). We hope that our analysis infrastructure helps to identify a sweet spot for exploiting the most relevant interaction characteristics, without the overhead of our current dynamic analysis implementation in VarexJ.

In several traces, we measured interactions of which not all might be intended. We conjecture that our dynamic analysis might be useful for developers to understand the sources of interactions and to build maintainable and assurable software.

4.5 Related Work

Despite much research on highly-configurable systems, the nature of configuration-related interactions is not well understood, especially at the code level. In studying bug reports, many studies found that the majority of reported configuration-related bugs are caused by individual options or interactions among only few options with only few defects at higher degrees [Abal et al., 2018, Cabral et al., 2010, Cohen et al.,

2007, Garvin and Cohen, 2011, Kuhn et al., 2004, Medeiros et al., 2016, Nie and Leung, 2011]; but none of these studies is based on a configuration-complete analysis. Manual search for feature interactions in requirements in telecommunications and electronic mail has focused primarily on pairwise interactions [Hall, 2005, Kolberg et al., 2000]. The few studies that systematically analyzed entire configuration spaces found also interactions among more options, such as a linker fault in Busybox that involved 11 options [Kästner et al., 2012a]. Where prior work primarily focused on the degree of interaction faults, we define and monitor measures for interaction degrees and interaction overhead to assess configuration complexity of both data and control flow interactions.

Halin et al. [2018] tested all 26,000+ configurations of the JHipster with the goal of evaluating the quality of sampling strategies and to analyze interaction faults. As they tested each configuration in individually, their experiments took 182 days (CPU time). They detected six interaction faults triggered by up to four options. In contrast, we used an efficient configuration complete analysis that allows us to scale executing test cases for large configuration spaces for which it would be infeasible to test them all. Variational execution further enables us to observe feature interactions beyond failures.

As discussed in Section 3.6, symbolic execution can potentially behave like variational execution [Baldoni et al., 2018, Sen et al., 2015]. Thus, symbolic execution could be used to perform a similar analysis of feature interactions. However, scaling symbolic execution to large systems remains challenging due to expensive SMT solving. In contrast, variational execution has much lower effort as computations are reduced to concrete operations and SAT solving [Meinicke et al., 2016, Nguyen et al., 2014b, Wong et al., 2018b].

Closest to our analysis of interactions at the execution level, Reisner et al. [2010] used symbolic execution to explore different paths of test cases in three C programs (9–14 KLOC, 13–30 options) and found interactions among 7 of 30 options in one system. Executing configurations separately, they measured the effect of interactions on control flow only (with the goal of increasing test coverage), whereas we specifically monitor the effect of interactions on data to measure configuration complexity (to assess whether a configuration-complete approach is feasible), especially regarding the different notions of local variability (e.g., using benchmarks B3–B5 and the complexity measure of interaction overhead).

Nguyen et al. [2014b] previously used variational execution to observe interaction degrees on data finding that up to 16 plugins interact on the same data. In contrast, we systematically analyze how such interactions appear to get a better understanding of interaction characteristics.

4.6 Conclusion

Undesired interactions challenge quality assurance for highly-configurable software, as they are typically unknown and can result in faults and security vulnerabilities. Their detection is a challenge as the configuration space of such systems grows up to exponentially in the number of configuration options. Existing analyses try to

scale with assumptions about interactions. However, whether these assumptions are valid and how much we can speed up analyses in future is not well understood. With VarexJ, we implemented a dynamic analysis for Java to quantify different characteristics of interactions with benchmarks and to analyze real-world programs (see Chapter 3) [Meinicke, 2014]. We found that essential configuration complexity induced by real-world interactions is usually low, making configuration-complete analyses feasible. Based on our insights, we discussed typical characteristics of interactions. These characteristics are unknowingly exploited by resend analyses [Thüm et al., 2014], but can intentionally be exploited by future approaches, such as for debugging as we will show in Chapter 5 [Meinicke et al., 2018].

5. Understanding Interactions in Highly-Configurable Systems with Variational Traces

This chapter is based on the technical report "Understanding Differences among Executions with Variational Traces" [Meinicke et al., 2018].

Understanding why a certain program input causes a fault while another succeeds is a common task during debugging [Zeller, 1999]. This happens, for example, if a certain program crashes in one configuration but succeeds in others [Cohen et al., 2007, Medeiros et al., 2016, Nie and Leung, 2011]. Reasoning about such differences in executions is difficult when using a standard debugger as the program can only be executed for one input at a time. To understand why certain inputs lead to a fault requires to understand the differences between correct and failing executions. Such a comparison can be generated by recording and aligning the traces and state changes of these two executions [Sumner and Zhang, 2010, Xin et al., 2008]. The aligned traces can be used to generate explanations of the fault [Sumner and Zhang, 2013, Zeller, 2002].

Some faults, however, are caused by interactions of multiple inputs which make understanding and debugging them even more challenging [Abal et al., 2018, Apel et al., 2013b, Calder et al., 2003a, Garvin and Cohen, 2011] (see Chapter 2). Interaction faults are hard to detect as they require to specify a certain input to trigger the fault [Medeiros et al., 2015]. Even if we can narrow down the fault to a smaller number of options, say with *delta debugging* [Zeller, 1999, Zeller and Hildebrandt, 2002], it is still difficult to understand *why*, *where*, and *how* they interact.

After identifying the set of interacting options, a programmer can start investigating how this interaction causes the fault. Understanding the interaction requires understanding the individual behavior of the interacting options, but also their combined behavior. Thus, it may no longer be sufficient to align the execution of two inputs as previous approaches do [Sumner and Zhang, 2013, Zeller, 2002], as such an alignment cannot explain the effects of multiple options.

We propose to align the execution traces of *all* configurations to explain the effect of multiple options. We introduce the concept of a *variational trace*, a compact representation of the trace differences among all executions. In the variational trace, redundant parts are shared, and individual parts are annotated with the input they belong to. This focus on differences allows understanding how data and control flow influence the executions and interact, and thus how the different inputs cause a fault.

Generation of variational traces challenges scalability as the number of executions and traces that need to be aligned can potentially be exponential to the number of options. A baseline approach would execute all configurations separately and thus can only scale to explain the interactions among few options. More severely is the potential memory consumption as this approach needs to keep all past statements in memory, as it is unclear upfront which statements will differ among executions [Ko and Myers, 2010, Pothier and Tanter, 2009].

We use variational execution [Meinicke, 2014, Meinicke et al., 2016, Nguyen et al., 2014b, Wong et al., 2018b] to avoid separately recording and aligning the traces for many inputs (see Chapter 3). Since the program is executed in a shared fashion, alignment is achieved by construction. Additionally, executed statements and program states are tracked as they relate to the original inputs and their interactions. This means that executions and variables can always be linked to specific program inputs. With variational execution we usually avoid the memory explosion as it enables us to recognize during execution which statements differ, and thus, need to be kept for the variational trace. As options are mostly orthogonal and appear only on few locations we expect variational traces to be concise [Meinicke et al., 2016] (see Chapter 4).

To enable developers to interact with variational traces, we developed an interactive Eclipse plug-in called *Varviz* that visualizes variational traces. Varviz can be used for understanding interaction faults, but also for other program comprehension tasks that involve understanding differences among similar executions.

In a user study, we show that variational traces help to understand a fault in a highly interacting program and that they help to explain a fault from a previous experiment on automatic debugging techniques. Participants using variational traces were more than twice as fast and more successful compared to participants using a standard debugger. In our qualitative study, we found that when dealing with faults caused by differences in inputs, in contrast to standard faults, participants search for places where these inputs interact and how they have an effect, which is exactly what our approach helps with. Furthermore, we show that variational traces are compact, even for medium sized programs with many explored options.

The goal of this research is to aid researchers and developers detecting differences among executions depending on potentially multiple options by providing a dynamic analysis that can efficiently execute and align all configurations summarized in variational traces.

Overall, we contribute the following in this chapter:

- The concept of a *variational trace* that compactly represents differences among the executions of many configurations.
- A *baseline implementation* that shows challenges of generating variational traces.
- *Efficient trace alignment* and a *dynamic analysis to trace only relevant data* avoiding memory explosion for a potentially exponential number of configurations using variational execution [Meinicke, 2014, Meinicke et al., 2016].
- An Eclipse plug-in *Varviz* to visualize and interact with variational traces.
- A *user study* that shows that participants using variational traces outperform participants using a standard debugger. The study also shows that comparative approaches [Sumner and Zhang, 2013, Zeller, 2002] actually help with debugging tasks.
- An *evaluation on scalability* showing that our approach can align executions for large number of input combinations, while concisely describing their differences. By focusing on the effects of certain inputs, the sizes can be reduced even further.

5.1 State of the Art

An important problem of many program comprehension and debugging tasks is to understand the differences among executions. For example, a programmer may want to understand why a fault occurs for certain inputs but does not for others. Such faults can be hard to detect, as they are only triggered for a certain input, and hard to understand because they may require reasoning about interactions among these inputs within long traces. The differences in control and data flow among faulty and valid executions can provide insights about how the fault is caused.

Our approach combines ideas from two research fields, namely *automatic debugging* and *feature interactions*, to explain differences among executions. In this section, we discuss how automatic-debugging techniques exploit differences in executions and how various approaches address the feature-interaction problem that causes undesired behavior for certain combinations of inputs.

5.1.1 Automated Debugging Techniques

Automated-debugging techniques aim to create explanations of why a fault appears, often by comparing correct and faulty executions of the program [Johnson et al., 2011, Jones et al., 2002, Sumner and Zhang, 2009, 2013, Weeratunge et al., 2010, Zeller, 2002]. To create explanations, the approaches execute, record and align the program multiple times for different inputs or test cases.

Fault Localization

There are many approaches that aim to find the cause of a fault [Wong et al., 2016]. One of these approaches is *spectrum-based fault localization* which rates each line of the source code by whether it is probable to cause the fault [Abreu et al., 2006, Jones

et al., 2002]. For example, Tarantula compares the code coverage of valid and failing test cases and provides this information using different background colors on the code [Jones et al., 2002]. However, such statement ranking has been shown to be less useful than expected [Parnin and Orso, 2011]: there are usually too many lines highlighted in the code, which makes it hard for users to understand which lines matter for an explanation of the fault. Also, the information why certain lines are important is missing, as are control- and data-flow information. Such information is however often necessary to understand how certain parts of the execution lead to the fault.

Execution Comparison

The comparison of internals of failing and valid executions can be used to explain why programs fail. Instead of just comparing the coverage information, *execution comparison* approaches compare traces and program states across executions [Apiwattanapong et al., 2007, Groce et al., 2006, Johnson et al., 2011, Kim et al., 2015, Sumner and Zhang, 2009, 2013, Weeratunge et al., 2010, Zeller, 2002]. Such comparison can highlight differences in data and control flow relevant to the fault.

Delta debugging is an approach that systematically narrows down the inputs (resp. changes) that are relevant for causing a fault [Zeller and Hildebrandt, 2002]. Based on this idea, Zeller applied delta debugging to program states of executions [Zeller, 2002]. A challenge with delta debugging is that it needs to align statements and program states of independent executions [Xin et al., 2008]. Sumner et al. [Sumner and Zhang, 2009, 2013] improved the initial work of Zeller using dual slicing [Johnson et al., 2011, Weeratunge et al., 2010] and efficient execution indexing [Xin et al., 2008]. They improve the efficiency and minimize the explanations in finding the cause effect chain [Sumner and Zhang, 2009, 2013].

Execution comparison approaches are designed to explain the differences between only two executions at a time. Thus, they require significant runtime overhead, as they execute the program many times to narrow down the instructions necessary for a certain behavior. Due to the separate execution of the program, these approaches need to deal with three major challenges, correct alignment of the executions [Kwon et al., 2016, Sumner and Zhang, 2010, Xin et al., 2008], memory overhead that comes with recording the executions [Burtscher et al., 2005, Kanev and Cohn, 2011, Ko and Myers, 2008], and differences in executions caused by nondeterminism [Kwon et al., 2016].

5.1.2 Understanding Feature Interactions

Feature interaction bugs are hard to detect as they are only triggered for certain combinations of features. There is a lot of research to efficiently find such faults, such as combinatorial interaction testing [Cohen et al., 2007, Medeiros et al., 2016, Nie and Leung, 2011], systematic sampling [Kim et al., 2010, 2013, Souto et al., 2017], model checking [Burch et al., 1990, Classen et al., 2011, von Rhein et al., 2011], and variational execution [Austin and Flanagan, 2012a, Kim et al., 2012, Meinicke et al., 2016, Nguyen et al., 2014b, Wong et al., 2018b].

Sampling approaches can only reveal configurations that fail, but not the interaction that causes it. Variational execution tracks the exact combination of inputs that

lead to a fault but does not help to understand *why* the interaction happens and *how* it causes the fault.

Other approaches statically reason about the code to detect feature interactions. The work of Kim et al. [2011] reasons about the combinatorics to reduce the number of configurations to execute. However, after testing these reduced configuration, the approach cannot answer *why* configurations fail. The research of Zhang and Ernst aims to identify configuration faults using thin slicing and a lightweight form of execution comparison [Zhang and Ernst, 2013, 2014]. Their approach suggests single configuration options that are likely to trigger a fault. However, they assume that (a) the program itself is correct and (b) that a single option triggers the fault rather than a combination of multiple options.

Another line of research aims to identify implementations of feature in configurable systems, known as *feature traceability problem* [Apel et al., 2013a]. Many solutions from software product line research can deal with the feature traceability problem for programs where feature can be directly mapped to code. In these programs, variability is either implemented in modules (e.g., using feature-oriented programming [Prehofer, 1997] or plugins) or variability is explicitly annotated (e.g., for conditional compilation). There are several approaches that help users deal with feature traceability in such systems using views or background colors [Feigenspan et al., 2011, Kästner and Apel, 2009, Meinicke et al., 2017]. These approaches, however, ignore data-flow and control-flow dependencies among the features.

In contrast to systems with a direct mapping of features to source code, feature traceability becomes challenging, such as for systems with runtime variability [Apel et al., 2013a]. Lillack et al. [2017] reason about which lines of code are affected by load-time options using static taint analysis [Arzt et al., 2014]. Nguyen et al. [2016] iteratively sample configurations and observe interactions on code coverage. Similarly, Reisner et al. [2010] observe interactions on code coverage using symbolic execution.

Feature traceability can help identifying features in source code and can help with code comprehension [Feigenspan et al., 2013]. Additional information about which features interact and which lines are affected by these interactions can provide further help for understanding and debugging. However, identifying features in source code and information on which features potentially interact cannot answer *why* and *how* they interact.

In summary, there exist many approaches that help to compare two executions and approaches that help detecting interaction faults in larger configuration spaces. However, none of the existing approaches helps to understand *how multiple options interact* as they either do not scale to multiple options or miss control and data-flow information [Ko and Myers, 2008, Ko et al., 2006b]. In our work, we provide support that scales to explain interactions among multiple options and that also provides necessary control and data-flow information about the interactions.

5.2 Generating and Visualizing Variational Traces

In this section, we introduce the new concept of *variational traces* which help developers understanding how options interact during the program execution and assist with debugging interaction faults.

5.2.1 Variational Traces

We introduce a *variational trace* as a compact representation of differences among multiple execution traces regarding control flow and the program states. With variational traces, we explain how differences in inputs affect a program's execution and data, and how inputs interact with each other. A variational trace is a graph that represents differences on the control-flow and on data using the following concepts (illustrated in Figure 5.1 for our running example GameScreen of Listing 2.1):

- *State changes* (orange rectangles) describe statements that change values of fields and local variables. In the example, the value of `penalty` is changed from *0* to 10 if the option *blue* is true.
- *Decisions* (diamonds) describe statements causing control-flow differences due to differences in inputs (directly or indirectly) between the individual traces (e.g., if-statements).
- *Decision Parameters* (gray rectangles) describe variables that are used in decisions. In the example, `score` is used as parameter for the decision in the method `setScore`. We found this information on parameters used in decisions particularly useful as they help to understand causes of control-flow differences.
- *Exceptions* (red rectangles) describe statements that are thrown due to faults or exception handling. In the figure, we see that the `RuntimeException` is thrown under the condition that *blue* and *red* are true.
- *Return* statements (not shown in the example) describe values that are returned by a method. Return statements are included if a method returns different values or from different locations.
- *Methods* (rectangles around subgraphs) structure the variational trace and describe the stack trace (e.g., method `setScore`). The notation of methods helps to understand the control flow and the execution of the program across method boundaries.
- *Paths* connect the elements and show the control flow. Paths are annotated with *path conditions* showing under which configurations the path is taken (e.g., $blue \wedge red$). Additionally, the failing path is highlighted: *red* if the path always leads to a fault, and *orange* if the path eventually leads to a fault. Orange paths are bold if they are necessary to cause the fault, they are dotted otherwise.

Variational traces show differences

To be a useful debugging tool, variational traces need to concisely describe differences among executions while still containing sufficient information. A variational trace only describes state changes, decisions, invocations and exceptions that differ among executions. Thus, a variational trace contains only statements that cause control-flow differences and statements that change variables differently in different executions. Statements that do not cause such differences are not as important and will not be contained in the variational trace. For example, initialization of fields in

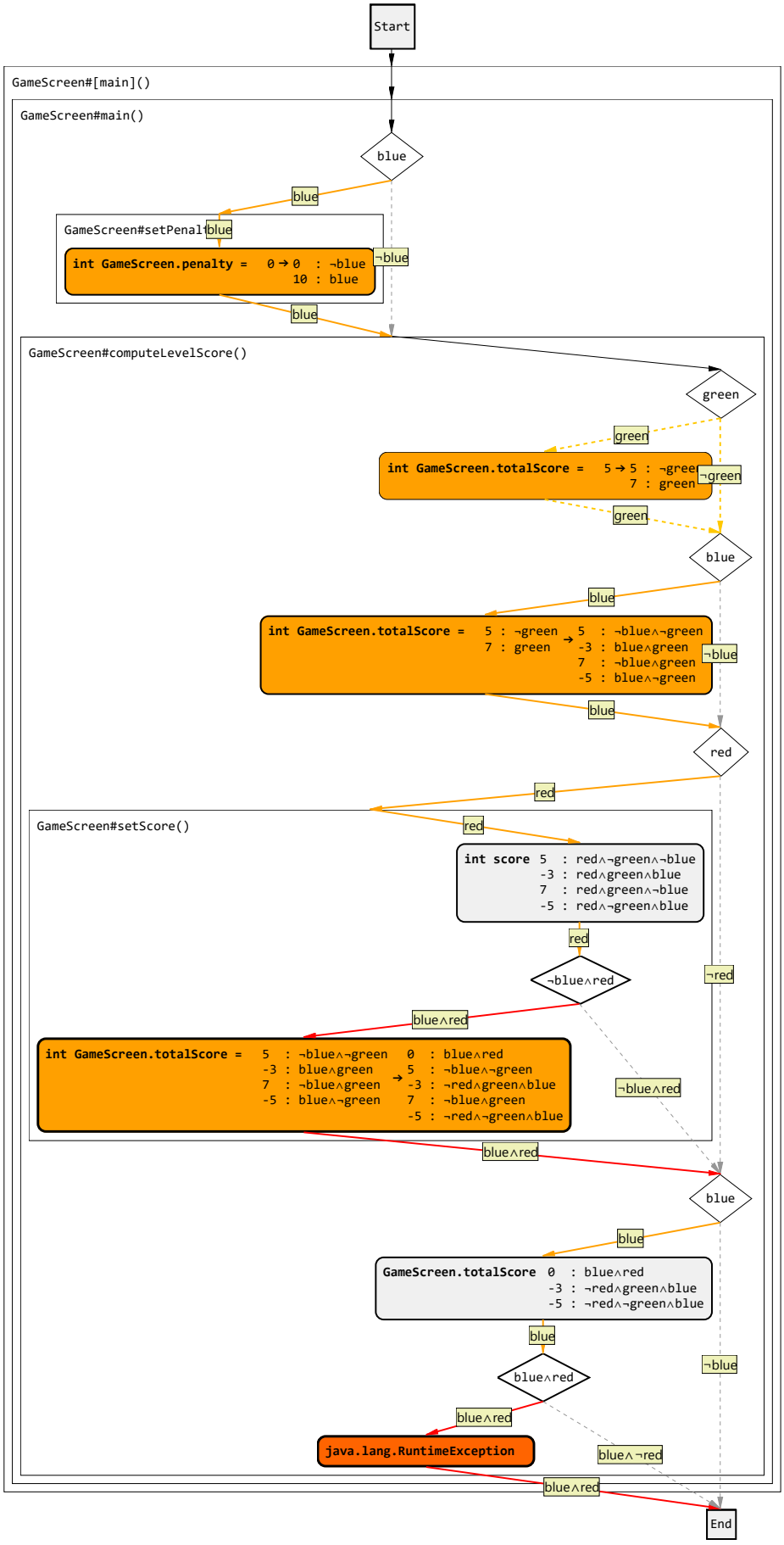


Figure 5.1: Variational Trace for our running example GameScreen.

the constructor of the GameScreen are the same for all configurations. Such state changes are thus not relevant to describe differences among executions. The same applies to changes that only take place on variables that only exist for certain inputs. Thus, if an object only exists under condition *a*, then changes that happen to this object under condition *a* are not important for comparing traces. Instead, the variational trace will report changes that create interactions, such as the method `setPenalty` which sets the field `penalty` to 10 if *blue* is true. Such a statement reduction requires to record and keep all state changes of all variables and to compare their values across all executions.

A focus on differences among executions allows to narrow down the number of statements that are reported to the user. Our approach might be further combined with slicing to remove statements that do not matter to explain a certain exception [Sumner and Zhang, 2009, 2013, Zeller, 2002]. In this work, we focus on efficiently generating explanations for differences among many executions. Thus, improvements such as data-flow and impact analyses are out of this paper's scope.

A variational trace, as in Figure Figure 5.1, looks similar to a control flow graph. However, three differences make variational traces more useful for debugging purposes: (i) variational traces represent actual executions and they show the actual values that variables take plus the exact context they are executed in, (ii) variational traces only contain statements that differ among the executions, and (iii) variational traces support (beyond others) loops, recursion, dynamic invocation, and reflection.

Using Variational Traces

Variational traces help understanding differences among executions as they describe the cause and effects of differences among executions. This allows answering questions about interactions among options and to understand faults caused by such interactions. First, a variational trace answers for which inputs the interactions occurs. This information is visible in the context of the statement (resp. exception). Second, it helps understanding why the interaction occurs as it shows the differences in the state and in the control-flow at the point in the execution before the statement. Third, a variational trace helps understanding how the interaction is caused and how the corresponding state is created. As an interaction is always caused due to previous control-flow decisions or other interactions, the information about the cause will always be contained in the variational trace.

5.2.2 Generating Variational Traces by Aligning Trace Logs

To generate variational traces, we could align the traces from executing all configurations separately. In this subsection, we use such a base line approach that aligns trace logs to explain the necessary steps and emphasize the challenges of creating variational traces, namely recording, alignment, merging, and reduction.

The first step for creating a variational trace is to *record* the program's executions for *all* different inputs. The recording needs to track both the instructions and the state changes, to understand the effects of the executions.

To compare the executions, the recorded traces need to be aligned. To *align* the traces, instructions from each trace that belong to each other need to be identified [Needleman and Wunsch, 1970]. In general, there are multiple different alignments possible as statements may repeat in different parts of the execution. Thus, a semantically optimal alignment is required for meaningful results. However, as the program is executed separately multiple times, alignment is a non-trivial task, especially as object references are different among executions, as it is necessary to keep track of iterations, and as nondeterminism may lead to wrong alignments, which can be avoided by recording and replaying nondeterministic result (e.g., IO) among the executions [Kwon et al., 2016, Sumner and Zhang, 2010, Xin et al., 2008].

After aligning the traces, shared parts of the executions can be identified. Thus, we can *merge* the traces into a single trace with conditional statements [Rubin and Chechik, 2013]. Statements and state changes that are included in several traces can be shared.

Finally, the variational trace can be *reduced* to statements that describe differences and can be enriched with information about the effects of the instructions. The variational trace only describes the control-flow and state differences among the executions resulting in a concise explanation of the differences and the fault.

Discussion

The baseline approach has the obvious issue that it can only scale to few options as it needs to run the program 2^n times for n boolean options. Furthermore, as it is unclear which statements will interact upfront, the approach needs to record full traces which causes severe memory problems. That is, the approach needs to keep all statements and states in memory until all configurations are executed. There are several ways to reduce the memory consumption, such as (i) directly merging the traces after execution instead of keeping them all in memory, (ii) statically deciding which parts of a method will be equal independent of options (e.g., initialization of local variables), (iii) compressed storage of trace logs [Burtscher et al., 2005, Kanev and Cohn, 2011], and (iv) inter-procedural analysis to detect statements that depend on options [Lillack et al., 2017]. However, these approaches are either not sufficient (i, ii & iii) or do not scale to larger programs (iv).

We implemented the baseline approach in a prototype for Java including optimizations (i) and (ii). We observed that the additional logging statements and the merging process cause a lot of computational overhead. However, especially the memory consumption of the approach is problematic. If too many statements and states must be kept in memory the tool may record multiple GB for the systems in our evaluation, as we show in Table 5.1, which makes this approach infeasible for larger systems. To approach these challenges requires sophisticated mechanisms to reduce the number of statements to record and to solve the problems of nondeterminism (e.g., by synchronously executing all configurations [Kim et al., 2015, Kwon et al., 2016]).

Our goal is to generate variational traces anyhow as they enable us to observe effects and interactions of options. To this end, we have to avoid all the limitations (scalability, memory overhead, nondeterminism) of the base line approach while keeping the idea of aligning the executions of all configurations.

5.2.3 Efficient Generation of Variational Traces with Variational Execution

A key insight of our work is to use *variational execution* to sidestep the previously discussed challenges for generating variational traces (i.e., handling the challenges of recording, alignment, merging and nondeterminism). Variational execution can efficiently execute *all* configurations of the program in a single synchronized run [Meinicke et al., 2016, Nguyen et al., 2014b]. As variational execution represents state differences among configurations as choice values [Erwig and Walkingshaw, 2011a], we can efficiently observe the states of all configurations [Meinicke, 2014, Meinicke et al., 2016]. We solve the problem of non-deterministic behavior among the configurations due to the sharing and synchronous execution of variational execution, for which an alignment-based approach requires sophisticated synchronization and alignment strategies [Kwon et al., 2016, Sumner and Zhang, 2010].

As variational execution synchronizes the executions among configurations, there is no need to align the executions of the program. Instead, we directly generate the variational trace by recording how variational execution runs the program. The context of each statement is already given as all instructions are executed under a certain context. To observe where the execution splits, we just need to observe changes in the execution context. Finally, we can observe interactions on data in the assigned choice values of local variables and fields. With variational execution we avoid the memory explosion of the base line approach, as we can decide during execution whether a statement should be contained in the variational trace. As we will show as part of our evaluation in Table 5.1, using variational execution is up to five times faster than the baseline approach while requiring up to 74% less memory. Beyond that, we also show that we can generate variational traces for huge configuration spaces for which it is simply impossible to generate them using a brute force approach (see Checkstyle).

To generate variational traces, we extended our variational interpreter VarexJ [Meinicke, 2014, Meinicke et al., 2016]. We adapted the execution of bytecode instructions, such as fields and local variable instructions to record their changes on data, if-statements to record whether they split the execution, method invocations to record the stack trace, and exceptions to report faults.

Our new contribution is to realize the potential of variational execution for generating variational traces without the disadvantages (e.g., memory overhead) of aligning single traces, by observing internals of the variational execution engine. Thus, with variational traces, we are usually able to align an exponential number of traces, which is practically impossible without it.

Generation with Symbolic Execution

Variational execution shares similar ideas with *symbolic execution* [Clarke, 1976, King, 1976] (see Chapter 3). Indeed, with symbolic execution it is possible to explore the executions for many inputs. In contrast to symbolic execution, variational execution always processes alternative but concrete values, not symbolic ones. Thus, variational execution does not share the problems of symbolic execution, such as undecidable loop bounds. As variational execution requires concrete inputs, our

approach also requires a test case, which is given by our scenario as we want to compare executions for a given test. Symbolic execution can hardly be used to generate variational traces as it typically does not share and align executions beyond common prefixes [Cadar et al., 2011, Hentschel et al., 2016b].

As discussed in Section 3.6, symbolic execution engines that implement a fully symbolic memory, if-then-else formulas, and optimal joining of execution branches can potentially have similar sharing as variational execution [Baldoni et al., 2018, Sen et al., 2015]. Such symbolic execution engines can potentially be used to generate variational traces as the execution may be similar to variational execution and the data-flow differences can be observed in the summary values. However, due to the overhead of constraint solving, it remains unclear whether they would achieve the same scalability and performance as variational execution.

5.2.4 Varviz

We argue that variational traces aid programmers to debug and understand programs by providing information about the program execution and interactions among options. To make variational traces accessible, we implemented an Eclipse plug-in called *Varviz* (from *var*iation and *vis*ualization). We released Varviz as open-source (https://meinicke.github.io/varviz/). The plug-in already comes with the utilized VarexJ to generate variational traces.

In Figure 5.2, we show a screenshot of Varviz for our running example GeamScreen. The variational trace can be generated using default run mechanisms of Eclipse, which will automatically call VarexJ. After running the program, the variational trace is shown in the Varviz view.

Navigation is one of the most time-consuming tasks during debugging [Ko et al., 2006a]. Therefore, we designed Varviz to also be used as a navigation tool. By double-clicking on elements in the trace, the tool automatically displays the file and the line of the element. As shown in the screenshot, the return instruction that throws the exception is highlighted after double-clicking the exception statement in the trace.

Focus on selected interactions

In practice, many interaction faults occur among few options [Abal et al., 2018, Cabral et al., 2010, Cohen et al., 2007, Garvin and Cohen, 2011, Kuhn et al., 2004, Medeiros et al., 2016, Meinicke et al., 2016, Nie and Leung, 2011]. To understand a certain interaction among a specific set of options, we show a, usually smaller, relevant part of the variational trace. To this end, we provide a new *projection mechanism* to show only the statements that explain the effects of a given set of options. All other options are set to fixed values, false if possible (the valid selection might be restricted by a feature model [Kang et al., 1990]). By setting all other options to fixed values, Varviz will produce a *projection* for the interaction of the few options of interest. For example, if a fault is thrown under context $A \wedge \neg B$, we are interested in the interactions of these two options, but not in options C and D. To create a projection on the variational trace for A and B, we set the other options C and D to false (if possible) to hide the effects and interactions of these

options. To remove unnecessary elements, we create the constraint $\neg C \wedge \neg D$ and evaluate the conditions of all elements of the variational trace. If the condition of the element under the context of the constraint is satisfiable, we keep the element in the variational trace, otherwise it can be removed. Finally, we evaluate the remaining elements, whether they represent differences among the executions for the options of the projection. Removing options from the trace can highly reduce its size and thus helps to understand the interactions (as we will show in Section 5.5), while preserving interactions that are relevant for the options of interest.

5.2.5 Limitations of Variational Traces

Variational traces inherit limitations from related automatic debugging techniques based on trace alignment [Sumner and Zhang, 2013, Zeller, 2002]. Similar to these techniques, we compare separate executions to explain causes of faults and interactions. The similarity of these executions determines the quality of variational traces. For example, executions (i.e., test cases) can introduce minor changes (noise) that are irrelevant to the fault of interest. Thus, executions that minimize noise are always preferable. In contrast, inputs need to trigger similar executions to reveal enough information about fault and its cause. If the executions are too different, then the variational trace cannot provide enough information. When we apply variational traces to configurable systems for the same test case, the executions of the configurations will be similar by design. However, if a fault is not caused by an interaction, but simply because certain code is executed, then we can only report the context and location of the fault but not necessarily its cause.

We use our variational execution engine VarexJ to generate variational traces [Meinicke, 2014, Meinicke et al., 2016]. Thus, we inherit the limitations of VarexJ, such as incomplete support for native code (see Section 3.4). Variational execution is an evolving technique and advancements in variational execution will also improve the efficiency and applicability of our approach [Wong et al., 2018b].

5.3 Variational Debugging of GameScreen

To illustrate how variational traces work on actual software, we again use our running example GameScreen. In Figure 5.2, we show the variational trace in Eclipse generated by Varviz (the full trace is shown in Figure 5.1). The trace explains how the options *blue* and *red* interact to cause the exception. As the fault is caused by the interaction $blue \wedge red$, we sliced trace is sliced the variational trace for *blue* and *red*. Thus, the option *green* does not appear in the variational trace as it is not relevant to explain the fault (i.e., *green* is set to be false).

To understand the fault, we can start at the exception on the bottom of the variational trace. We directly see that the exception is thrown under the context $blue \wedge red$. The next step is to understand why the exception is thrown under this condition. Thus, we check the decisions and parameters that lead to the exception. We can see that the parameter `totalScore` is 0 for $blue \wedge red$.

We can further backtrack the reason why `totalScore` is 0. Thus, we search for statements that change `totalScore` to 0 before. This leads us to the method `setScore`.

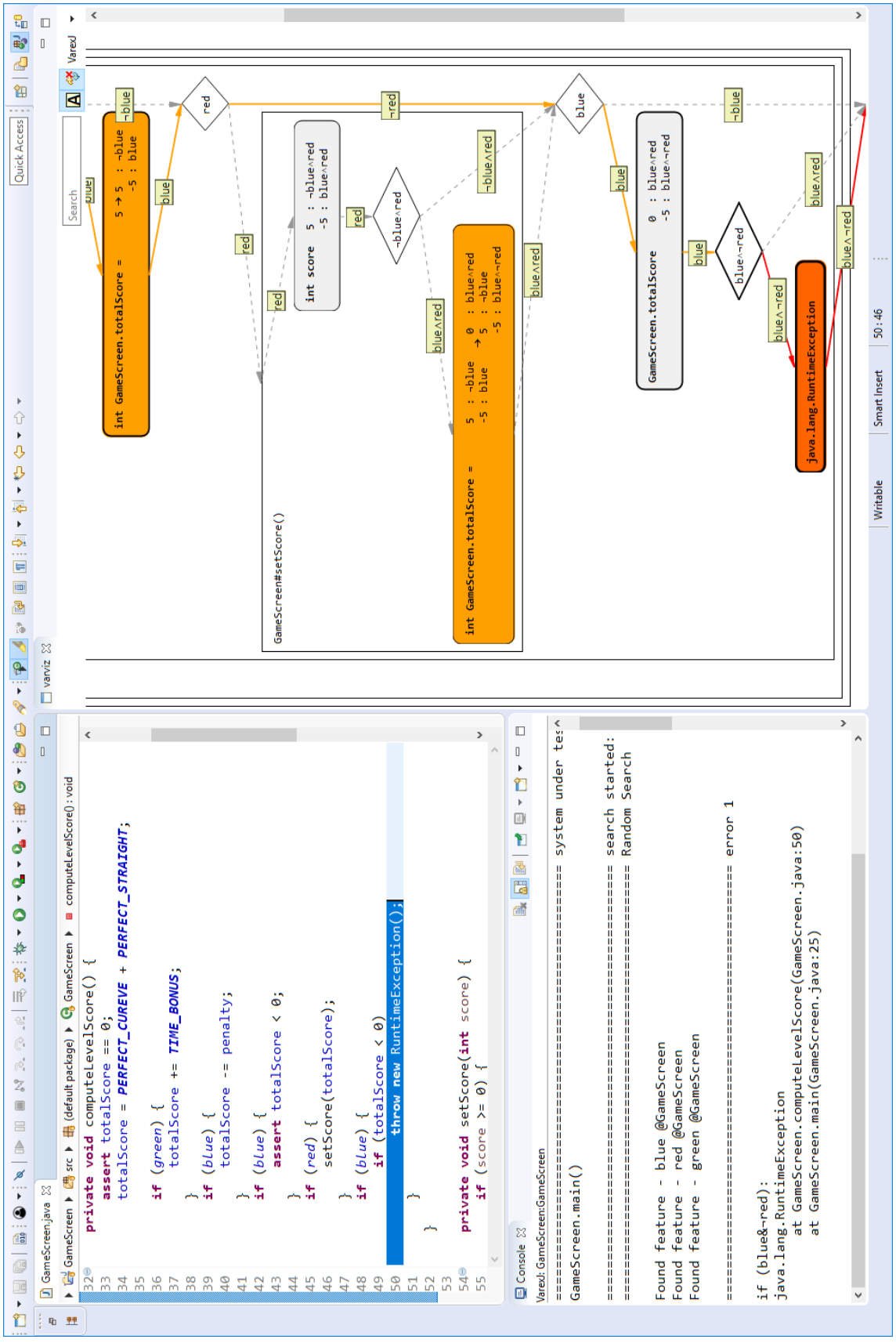


Figure 5.2: Screenshot of Varviz for our running example Gamescreen.

`setScore` changes the `totalScore` to 0 if the parameter `score` is negative, which it is if *blue* is true. We can also see that `setScore` is only invoked if *red* is true.

We further need to understand why `totalScore` was negative for `blue`. In the earlier statements we can see that under condition *blue* `totalScore` is changed to be negative because `penalty` was set to 10 under condition *blue*.

In summary, we can see that the variational trace contains all information necessary to understand how an interaction causes a fault. It contains information about control flow and and data changes caused by options. The slicing for the options that are responsible for the fault removes unnecessary options, which reduces the information on the user. As Varviz is integrated into Eclipse, it further provides a link to the source code which helps debugging.

5.4 User Study

Automated debugging techniques often promise large effects for debugging tasks. Previous evaluations on approaches based on execution comparison focused on reporting the size of the explanations (number of statements) instead of showing whether and how helpful they actually are for debugging [Sumner and Zhang, 2013, Zeller, 2002]. However, only reporting quantitative results of the approach can be misleading, and the expectations may not meet the reality (e.g., the explanations may be too complex and complicated to be understandable or do not contain the necessary information to understand the fault) [Parnin and Orso, 2011]. This is the first user study on delta-debugging-like approaches that we are aware of.

We designed variational traces to help users to understand variations in executions. In our evaluation, we investigate how and why variational traces help users. Specifically, we perform a user study to answer the following research questions:

RQ1: *How much do variational traces improve the performance of solving debugging tasks compared to a standard debugger?* To answering RQ1, we explore the speedup and the success rates for solving debugging tasks.

RQ2: *How do variational traces help understanding differences in executions?* With RQ2, we investigate what the information needs are during a debugging task and whether the variational trace can answer them. We want to evaluate whether the provided information (i.e, the statements shown in the variational trace) is sufficient to help understanding the interactions.

Systems and Tasks

We use three subject systems in our evaluation, namely GameScreen, Elevator and NanoXML. Statistics on the programs are shown in Figure 5.3. We carefully chose these systems for different reasons:

The first system is our running example *GameScreen* as it was previously used in a study which conducted the effect of different degrees of variability on program comprehension [Melo et al., 2016]. Melo et al. [2016] have shown that the small program with only three features takes on average ten minutes to debug without tool support. However, the program is too trivial and cannot give any insights for our

Program	LOC	Cov.	Opt.	Conf.	M	N	D	Instr.
GameScreen	32	32	3	8	4	12	6	230
Elevator	259	193	6	20	7	12	3	5,688
NanoXML	1000	331	1	2	18	21	4	42,138

Figure 5.3: Statistics on the programs used in the user study (Cov: Covered LOC by the test case, Instr: instructions executed for the test case, M: Methods, N: Nodes, D: Decisions).

study as the program can be understood in few minutes using a standard debugger. Thus, we used GameScreen only as warm-up task to make the users familiar with the type of tasks they should perform.

Elevator is a simulation of a configurable elevator system [Plath and Ryan, 2001]. The program is designed to trigger interactions among its options. Even though the program has only few lines of code, it is hard to understand the impact of its features due to the interactions. The program comes with several specifications in form of runtime assertions that are violated for certain configurations. We selected a specification that states that the elevator should continue in its current direction if there are still calls in this direction. This specification is violated if a feature for *executive floors* is on, which can force the elevator to change its direction. In the tasks for Elevator, the participants should figure out in which configurations the fault appears and how the fault is caused. Although fixing a fault is part of debugging, fixing itself is not part of the task as this would have required to change the program's specification. Instead it was sufficient to explain how the fault is caused.

NanoXML itself is not a configurable system. The program was used in a prior study to evaluate whether the automatic debugging technique Tarantula can help programmers with debugging [Parnin and Orso, 2011]. Tarantula showed only minor improvements for debugging NanoXML compared to a standard debugger. We evaluate NanoXML on the same bug as in the original study, in line with the original study [Parnin and Orso, 2011]. We provide two slightly different files as input for parsing. One of the files cannot be parsed correctly, causing an exception. The other one is a similar file that can be parsed. Both files are parsed simultaneously using variational execution. In addition to the tasks of the previous programs, the participants were also asked to fix the bug similar to the prior study [Parnin and Orso, 2011]. With NanoXML we show that variational traces are helpful for a standard debugging tasks to understand variations beyond configurable systems. Thus, with the NanoXML experiment we can show the usefulness of comparative- and delta-debugging approaches which have not been evaluated in user studies before [Sumner and Zhang, 2013, Zeller, 2002].

Pilot Study

We performed a pilot study to estimate the required power of our study (i.e., number of participants) to gain confidence finding significant results and to tune the task and descriptions. We asked several graduate students to use Varviz and the Eclipse

debugger on our tasks. We found and revised several issues with the usability of Varviz. We also measured the time and estimated that the effect size was big enough to show significant effects with few participants in the actual experiment. For the Elevator task we had an estimation of 40 minutes when using the Eclipse debugger, compared to an estimation of 12 minutes when using Varviz. For the NanoXML task we have an estimation of 22 minutes from a previous study when using a debugger, which we use as estimation when using a standard debugger (our results were slightly higher) [Parnin and Orso, 2011]. We have an estimation based on our pilot study of only 8 minutes when using Varviz.

Study Design

We designed our experiment as between-subject study to compare performances between participants using a standard debugger (baseline) or Varviz (treatment). We did not mix the participants between using the standard debugger and Varviz (i.e., within-subject design). Each participant solved all three tasks with the same tool to reduce training time required for Varviz and to avoid carryover effects, such as learning effects and demand effects [Charness et al., 2012]. Learning effects from the first tasks might be applied to the second which influences the performances when using different tools. Also, the motivation of using a new tool can influence the performance of the participants. This effect is amplified if they are using both tools in a within-subject design [Charness et al., 2012]. A between-subject design has less statistical power compared to a within-subject design (i.e., we may need more participants to show significant effects), however, we expect the effect size to be very large. Hence, as suggested by the literature on user studies [Charness et al., 2012], a between-subject design is more appropriate as it avoids confounding factors of within-subject designs.

We did not design two comparable tasks, but intentionally two very different ones for external validity. A within subject design typically requires multiple similar tasks, which is a benefit using a between-subject design. While the participants worked on the tasks, with their consent, we recorded the screen and asked them to verbalize their thoughts using think-aloud protocols [Beyer and Holtzblatt, 1997]. These recordings help us to track the participants' information needs and debugging strategies.

Other approaches, such as delta debugging [Zeller, 2002] and comparative causality [Sumner and Zhang, 2013] may give similar textual explanations of the faults. However, we cannot compare our approach with delta debugging [Zeller, 2002] and comparative causality [Sumner and Zhang, 2013] as the tools are not available (we contacted the authors) and as they are designed to explain differences among only two executions.

Methods

To answer RQ 1, we compare the time and success rates of the participants for solving the tasks. To answer RQ 2, we record the audio and the screen of the participants. We analyze the recordings based on qualitative content analysis using open coding [Saldaña, 2015, Schreier, 2012]. We watch the videos with the goal to

find common tasks that the participants perform during debugging. We use these commonalities to create a coding frame that allows us to understand how the participants perform when using Varviz or the standard debugger.

Participants

As we plan to perform think-aloud protocols, analyzing the data (i.e., screen recordings and audio) requires high effort. We thus aim to avoid an unnecessary high number of participants. According to Nielsen [1994], for think aloud protocols five participants are sufficient to gain most insights. Adding more participants does not give more essential information. Thus, to answer RQ 2, we require at least five participants per group, so ten participants in total.

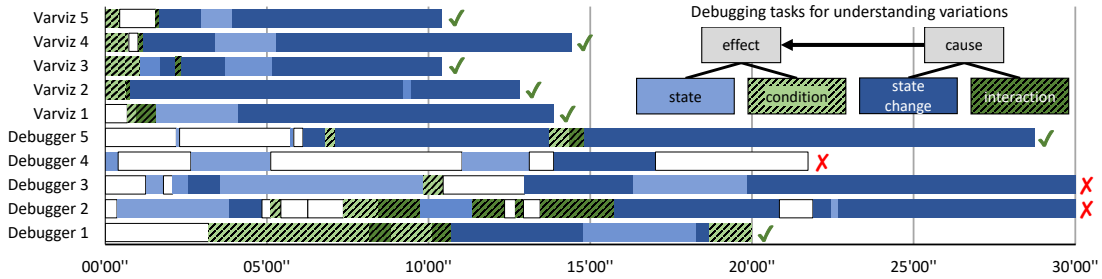
To answer RQ 1, we calculate the minimum number of number of participants required using power statistics based on the pre-study results. We use Rosner's equation to calculate the required sample size n for each group in our study [Rosner, 2015]:

$$n = \frac{(\sigma_1^2 + \sigma_2^2)(z_{1-\alpha/2} + z_{1-\beta})^2}{\Delta^2} \tag{5.1}$$

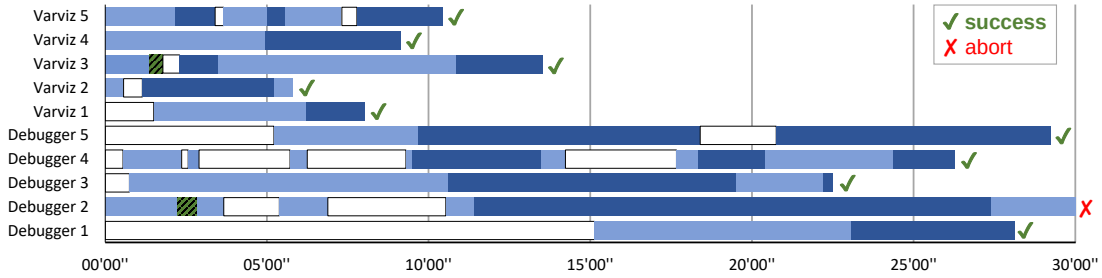
We use a conservative α value of 0.01 which is a probability of Type I error of 1% (i.e. the probability to find an effect even though there is none, usually 0.05). We use a conservative β value of 0.05 which is a probability of Type II error of 5% (i.e. the probability not finding an effect even though there is one, usually 0.2). The statistical power is $1 - \beta$ and thus 95%. We use a high value for the estimated standard deviations σ_1 and σ_2 of 5 minutes as we only used few participants in our pilot study. Using Equation 5.1 and our estimations of expected performances (10 versus 40 and 8 versus 22 minutes for Elevator and NanoXML respectively), we can calculate the number of participants needed for our experiment as one participant ($n_{elevator} = 0.989 \approx 1$) and five participants ($n_{NanoXML} = 4.544 \approx 5$) per group for Elevator and NanoXML respectively. As the required group size for NanoXML is larger than for Elevator, we use a group size of five for our experiment. Thus, based on our pre-study results, we need ten participants to show significant results. For both, our quantitative and our qualitative analysis ten participants are sufficient.

We recruited ten participants at Carnegie Mellon University: eight undergraduate students, one graduate student and one post doc. The participants were recruited using posters and mailing lists. We excluded participants without prior knowledge of Java. All participants received a 25-dollar gift card after finishing the experiment. The participants were assigned randomly to two groups of five. One group used the Eclipse debugger, the other group used Varviz. The graduate student used Varviz, while the post doc used the Eclipse debugger.

Before conducting the experiment, we asked the participants for their programming experience and experience with Java. The groups were roughly similar: The median experience for programming is 3.5 years for both groups (average is 3.7 years for the debugger group and 5.3 years for the Varviz group; note that the large difference in average experience is caused by a single outlier (Varviz 1) who reported 12 years of non-professional programming experience.) The median Java experience is 2 years



(a) Elevator



(b) NanoXML

Figure 5.4: Time spend on debugging tasks. White boxes denote unrelated tasks.

for both groups (average is 2.6 years for the debugger group and 2 for the Varviz group). None of the participants knew variational execution or any of the subject programs. All participants in the debugger group have used Eclipse and the debugger before. If a participant did not remember how to get to a certain view, such as the call hierarchy, or were unsure about certain functionalities of Eclipse, we provided this information during the experiment.

Execution

Both groups were given an Eclipse containing the three programs they had to debug including a failing configuration for Elevator and the two XML files for NanoXML. The participants using Varviz were introduced to the functionalities of the tool. We used a simple foo-bar example to explain the functionalities of Varviz. All participant started debugging the programs in the same order, first GameScreen, second Elevator and third NanoXML. The participants performed the tasks for Elevator and NanoXML until they solved them correctly, until they gave up, or until they reached a time limit of 30 minutes per task, as we planned the experiment to take roughly one hour.

As we performed a think-aloud protocol, we conducted the experiment with each participant in isolation. To record the audio and screen, we used an inhouse recording tool from our university. For the first participant using the debugger (Debugger 1), we used a different open source recording tool. We lost the screen recording due to a fault in the recording tool. However, we kept the results and the audio recording from this participant in our study as he had a good performance compared to the others using the debugger.

Quantitative Analysis

To answer RQ1, we compare the time and success rates of the participants for solving the tasks. In Figure 5.4, we plot the performances of the participants. The signs ✓

and ✗ indicate correct and aborted solutions respectively. When using the Eclipse debugger only two out of five participants solved the Elevator task correctly and four out of five for NanoXML. In contrast, all participants using Varviz solved the tasks for both programs.

For Elevator the participants using Varviz took on average 12 minutes while the participants using the debugger took on average 28 minutes. The best performance using the debugger took 20 minutes, more than five minutes longer than the worst participant using Varviz. For NanoXML we see similar results. The Varviz group took on average 9 minutes. In contrast, the debugger group took on average 27 minutes (best was 22 minutes). Our results are in line with prior research which reported an average time of 22:30 minutes using only a debugger [Parnin and Orso, 2011]. To calculate the *effect size* of using Varviz compared to a standard Debugger, we use the corrected equation of *Hedges' g* which corrects for the upwards bias on small sample sizes:

$$g = \frac{\mu_1 - \mu_2}{\sqrt{\frac{\sigma_1^2 + \sigma_2^2}{2}}} \times \frac{N-3}{N-2.25} \times \sqrt{\frac{N-2}{N}} \tag{5.2}$$

A g value of 1 indicates that the groups differ by 1 standard deviation. In general, a g value larger than 0.8 indicates a large effect size. We calculated a g values of 3.6 for Elevator and 4.9 for NanoXML. Thus, g values indicate a huge effect of using Varviz compared to a standard Debugger for understanding differences among executions. The differences are statistically significant (Mann-Whitney U test: $p < 0.01$).

RQ1: *How much do variational traces improve the performance of solving debugging tasks compared to a standard debugger?*

The participants using Varviz are on average **55%**, respectively **65%** faster than participants using the Eclipse debugger for Elevator respectively NanoXML. The success rates when using the Eclipse debugger are **40%** for Elevator and **80%** for NanoXML. When using Varviz, the success rates are **100%** for both programs.

Qualitative Analysis

To answer RQ 2, we analyzed the audio and the screen recordings to understand how variational traces help understanding differences among executions. Our coding frame [Saldaña, 2015, Schreier, 2012] is shown in the right upper corner of 5.4a. Overall, all participants try to understand the relationship between *effect* (e.g., the fault) and its *cause* [LaToza et al., 2007]. The tasks for understanding effect and cause can be refined further.

To understand the effect, it is necessary to understand program state and the condition. The program state is necessary to know values of variables and which method calls are important. When analyzing variations, it is also important to know under which condition (i.e., selection of options) the fault appears.

After understanding the effect, it is possible to investigate its cause. It is necessary to understand the state changes and method calls that lead to the state of the effect.

As the effect only happens under certain conditions it is also necessary to investigate how specific selections of options cause the effect.

In Figure 5.4, we used our coding frame to illustrate on which tasks the participants were working on. Additionally, to the tasks in our coding frame, the participants also spend time with unrelated tasks, such as reading, scrolling, or investigating unrelated code [Ko et al., 2006a]. Unrelated tasks are shown with white boxes. We plot the tasks on a horizontal time axis. The group using Varviz performed the tasks much better than the group using the debugger. In the following, we investigate the reasons why the tasks are so difficult when using only a debugger. We explore which information are required to solve the tasks and how Varviz helps to gather them.

We can see in Figure 5.4 that the debugger group spend much more time on unrelated parts of the program and on tasks that do not lead to solving the problem. The main reason is that the participants read unrelated code of the programs. Another reason is that the participants give up on their current goal and try to get information from other places. When using the debugger, it is up to the programmer to find the places where to find information about the program. Thus, the participants were lost in the source code they did not know, which leads to confusion and reading of code. In contrast, when using Varviz, the participants had a guide that helps them to find the few locations in the code that are of interest.

When we analyze the performances for Elevator more closely (see Figure 5.4a), we see that the Varviz group took only little time to understand the variations. All the participants almost instantly figured out the condition of the fault (which is trivial using the variational trace as it is indicated by the context of the exception statement). Also, figuring out where the option affects the behavior of the program is simple using Varviz as there are only few place (few decisions) where the option affects the data flow. In contrast, when using the debugger, the first task is to figure out the condition of the fault. Without specialized tool support, this requires switching the options in the configuration and to re-execute the program. The time spent on this task depends on how many options the program has, how many options interact, and finally on luck or intuition as for participant *Debugger 5*. A simple tool that reports the condition of the faults (e.g., brute force) would help the debugger users and would improve their performances. However, such a tool alone is not sufficient as it solves only a small part of the problem.

After finding the condition of the fault, the participants still need to answer how the option triggers the effect. By searching where the option is used, the place can be found, however also unrelated usages and only direct usages are found. Thus, participants *Debugger 3* and *Debugger 4* did not even identify this part of the program.

Finally, we can see that the Varviz group spend little time to understand the state at the exception. This means that they can spend more time for understanding how the interaction triggers the effect and how this causes the fault. The debugger group took overall more time to find the values of the variables and their values at the exception (except of participant *Debugger 5* who performed well on this task). The tasks to identify the exception state and the state changes are particularly hard as the program calls the scheduling method for the elevator in a loop. This makes

setting breakpoints hard as they are triggered multiple times before the actual state of interest.

In the performances for NanoXML (see Figure 5.4b), we can see that both groups struggle for identifying the state of the exception and answering why this state causes the fault. This is because of a relatively difficult if-statement shown in the listing below. The variational trace provides the values used in this if-statement. However, the participants still need to understand the meaning of it.

```
1 if (!str.equals(prefix==null?name:prefix+name))
2   XMLUtil.errorWrongClosingTag(this.reader, name, str);
```

After the participants using Varviz understood the meaning of the if-statement, they only spend little time to find the place of the cause as it is pointed out by a decision in the trace. In contrast, the debugger group again struggled to identify the cause. One reason is that the parsing is implemented using recursion, which again means that the breakpoints are triggered multiple times at the wrong state. This again shows that control-flow (i.e., the decision of the cause) and data-flow information (i.e., the values that differ among the two executions) are essential to understand the differences among executions. Both information are contained in the variational trace.

RQ2: *How do variational traces help understanding differences in executions?*

Understanding where and how configuration options influence the execution is hard using only a standard debugger as only one configuration can be executed at a time. Thus, participants using the Eclipse debugger have difficulties figuring out the **condition** of the fault, gathering information about differences in the **program states**, and to find the **cause** of the fault. Variational traces help with these task by providing essential information about the **fault condition**, the **fault state**, as well as **data and control-flow differences** that lead to the fault. This information helps users to focus on important parts of the execution which additionally prevents from wasting time on unrelated activities.

Threats to Validity

We designed our user study as between-subject study with only ten participants. As discussed, we decided to use a between-subject design to avoid confounding factors and to reduce training time at the cost of the reduced power of the design [Charness et al., 2012]. We carefully calculated the required number of users upfront to minimize the effort for analyzing the think-aloud protocols and the screen recordings [Rosner, 2015]. The measured performances in the experiment approximately match our expected performances from our pilot study. Due to the large effect sizes a larger number of participants is unnecessary. The fact that our results are statistically significant confirms that the chance of a random error due to small sample size is small.

To reduce selection bias, we recruit participants in public channels and randomly assign them into two groups. The average programming experience varies by almost

two years between the groups, which is however caused by a single outlier (Varviz 1). Our results are robust to removing this outlier (i.e., our results remain statistically significant without this participant) [Larsson et al., 2014]. To avoid effects due to differences in programming experiences, we designed the tasks in a way that basic debugging experience is sufficient. Since we conduct the user study in the Eclipse environment, experience with the Eclipse toolchain is likely to affect performance of participants. We performed a warm-up task to familiarize the participants with the type of tasks and the programming environment. We argue that most common usages of Eclipse are straightforward to most developers, given that Eclipse is a standard and classic development tool for Java. In addition, we made it clear to the participants before study that we could provide immediate support for questions on Eclipse usage. However, participants rarely asked for help regarding Eclipse. Thus, we argue that Eclipse experience likely has only minor impact to the performance results. To minimize confounding factors, we implement Varviz in a way that is completely orthogonal with existing features of Eclipse. Moreover, our introduction to Varviz only covers the plugin itself, not including any other functionalities of Eclipse. The think-aloud protocol influences the time performance of the participants; however, it influences both groups equally and because the expected effect size is big we do not expect any systematic influences on the overall results.

We used two diverse systems for the debugging tasks. We showed that variational traces are useful to understand faults in systems with multiple options as well to compare two executions. However, readers should be careful when generalizing our results to other systems and tasks.

5.5 Scalability Evaluation

Variational traces are concise representations of differences among executions. To further reduce the size, we allow focusing on small sets of options discussed in Section 5.2.4. However, we do not yet apply any kind of impact analyses to reduce the size even further, as this is out of scope of this paper and as the sizes are already small enough especially for the programs used in Section 5.4. Variational traces are useful beyond debugging, as for example in our work on detecting behavioral feature interactions with feature interaction graphs which can deal with large variational traces [Soares et al., 2018]. In this section, we evaluate the size of variational traces when aligning the executions of exponentially large configuration spaces. Specifically, we answer the following research question:

RQ3: *How does the generation of variational traces using variational executions scale compared to a base line approach?* The exponential growth of configuration spaces with the number of options is challenging for both, execution and alignment of many configurations. By answering RQ 3, we investigate the scalability of using variational execution to generate variational traces with regard to runtime and memory consumption.

RQ4: *How large do variational traces get?* Information on data and control flow differences useful are beyond debugging (see Section 5.6). With RQ4, we investigate how complex variational traces get when applied to programs with different numbers

Program	SLOC	Opt.	Conf.	Instr.	Time	TimeBL	Memory	MemoryBL	D_{all}	D_3	S_{all}	S_3
CheckStyle	14,950	141	$> 2^{135}$	194M	209.8s	*364.8.8s	1984MB	*3269MB	5,989	*165.5*	290,477	*2022.7*
QuEval	3,109	23	940	6M	10.6s	50.3s	379MB	1498MB	699	26.9	4,152	110.0
GPL	662	15	146	17M	15.6s	18.9s	408MB	975MB	530	6.9	5,565	301.0
Elevator	730	6	20	24k	0.1s	0.2s	41MB	29MB	36	17.7	96	51.0
E-Mail	644	9	40	26k	0.2s	0.4s	28MB	49MB	53	9.7	129	17.5
Mine Pump	296	6	64	22k	0.1s	0.4s	37MB	49MB	10	7.6	14	10.9

Table 5.1: Statistics on programs used in quantitative evaluation. D_{all} and S_{all} state the number of decisions respectively statements of the full variational trace. D_3 and S_3 state the mean number of decisions respectively statements after filtering for three options. In TimeBL and MemoryBL we show the time and memory consumptions for the baseline implementation (*for CheckStyle we only executed the configurations for five options with the baseline approach).

of options and different sizes. We also want to find out how effective the filters for options are for reducing the size (see Section 5.2.4).

Experimental Setup

In our evaluation, we reuse six configurable systems from our previous study on feature interactions and essential configuration complexity (see Chapter 4) [Meinicke et al., 2016]. The systems are shown in Table 5.1. The systems are from different domains and show different interaction properties [Meinicke et al., 2016]. We execute each system for a corresponding standard scenario. Each system comes with a set of options that can be enabled and disabled, which results in large numbers of configurations for which we execute the program (see Table 5.1).

The experiments are performed on a Windows computer with 8 GB ram and an Intel i5 processor with 4 cores. To answer RQ 3 we collect the execution time and the memory consumption, we ran variational execution and the base line approach ten times and report the median value to avoid measurement errors.

The size of the variational trace depends on several factors. First, the length of the execution (see number of instructions in Table 5.1). It also depends on how the program implements variability and how the options interact. The fewer options interact in a program, the shorter the trace will be (statistics on the interactions in the systems are discussed by Meinicke et al. [2016]). In our evaluation, we collect metrics on how large variational traces can get for different implementations and executions.

Statements and decisions indicate the complexity of the variational trace. However, understanding a fault or an interaction usually only requires few options as the interactions degrees are usually low [Abal et al., 2018, Cabral et al., 2010, Cohen et al., 2007, Garvin and Cohen, 2011, Kuhn et al., 2004, Medeiros et al., 2016, Meinicke et al., 2016, Nie and Leung, 2011]. We use context filters as discussed in Section 5.2.4 to filter the variational trace for all combinations of three options (we filter the trace of CheckStyle only for *one* option due to the large number of options).

Results

In Table 5.1, we report the times and memory consumptions required to generate the variational trace. As shown, the time to generate the variational trace is always

lower than with the base line approach, especially for larger configuration spaces. For Checkstyle with 141 options, our approach takes 209.8 seconds, which is lower than what the base line approach takes when aligning the configurations for only five options. As expected, the memory consumption of the base line approach becomes problematic, especially when aligning longer execution traces. Again, the base line approach requires 3 GB for aligning only the traces for five options while our approach takes 2 GB when aligning the traces for *all* configurations. Note that the reported memory consumption of our approach also contains the memory overhead of VarexJ itself.

RQ3: *How does the generation of variational traces using variational executions scale compared to a base line approach?*

Generating variational traces using variational executions scales to exponentially large configuration spaces with regard to execution time and memory consumption. In contrast, the base line approach is only able to generate variational traces for smaller configuration spaces while taking more time and memory than our approach based on variational execution.

In Table 5.1, we report the sizes of complete variational traces and the mean sizes after applying filters for three options. Even though the interaction experiments (Elevator, E-Mail and Mine Pump) are designed to cause many interactions among options, we see that the sizes of their variational traces are small. For GPL the complete variational trace becomes large as the executions contain long iterations which cause trivial but repetitive interactions on data (e.g., optionally initializing the weight for all vertexes in a graph). The same happens for QuEval which also has trivial but repetitive executions. For CheckStyle, which has by far the longest executions, we see that the size of the complete variational trace is huge. The size of the variational trace is, however, small in relation to the configuration space and the number of executed instructions. Most of these statements in the variational trace for CheckStyle are again repetitive. After applying the filter for one option, we can see that the size can be reduced a lot. In general, the sizes of the traces become small after filtering them. Even the full traces can be useful as most of the shown statements and decisions are repetitive due to iterations.

RQ4: *How large do variational traces get?*

Variational traces are often small, but can become large, especially due to long iterations that repetitively create the same interactions on data. The size can be drastically reduced due to the filtering mechanisms, such as filtering for a small set of options.

Threats to Validity

To mitigate threats to external validity, we analyze different programs that show different kinds of interactions. We argue that variational traces are scalable (i.e.,

we can align the execution for an exponential configuration space while the number of nodes does not grow exponentially with the number of configurations) to most programs in the wild, because recent studies have shown that although there could be many options in a program, most options interact locally and thus interaction degrees are usually low [Abal et al., 2018, Cabral et al., 2010, Cohen et al., 2007, Garvin and Cohen, 2011, Kuhn et al., 2004, Medeiros et al., 2016, Meinicke et al., 2016, Nie and Leung, 2011].

5.6 Applications Beyond Debugging

Understanding differences among executions can be useful beyond debugging a fault as in our example systems (see Section 5.4). In Section 5.1, we discussed related work that is able to detect feature interaction faults, help with feature traceability, and map options to program outputs. As discussed, these approaches can detect dependencies and interactions among options, but they cannot answer *why* they are interacting. We discuss three possible applications of variational traces to help understanding how dependencies and interactions are caused, to is illustrate the potential of variational traces in concert with existing approaches.

Understanding the Impact of Load-Time Options

It is difficult to identify code that is affected by load-time options, especially due to implicit data flow. Previous work used static taint analysis or symbolic execution to detect code that depends on certain options [Lillack et al., 2017, Reisner et al., 2010]. Information (e.g., background colors) about which parts of the code depends on variability can help in the context of conditional compilation [Feigenspan et al., 2013]. However, when considering indirect dependencies due to data flow, as for load time variability, understanding becomes more difficult. We found that information about which lines depend of which options alone is not sufficient for understanding, as the cause of indirect dependencies are hard to understand [Zulfiqa, 2016].[1] Understanding these dependencies requires information about their causes due to indirect data and control flow [Ko et al., 2006b, LaToza et al., 2007, Parnin and Orso, 2011]. With variational traces we can help to understand why certain parts of the code depends on the selection of options as we show causes of differences in the control flow [Meinicke et al., 2018].

Understanding Information Flow

Previous work compared executions to detect information flow, either using a similar technique to variational execution [Austin and Flanagan, 2012b, Austin et al., 2013, Schmitz et al., 2016, Yang et al., 2016] or using multi execution [Devriese and Piessens, 2010, Kolbitsch et al., 2012, Kwon et al., 2016]. In this context of information flow, confidentiality levels (e.g., high and low) are used as options of the program. Aligning executions allows detecting potential information leaks by observing the conditional outputs. However, tracing back why data is leaked is difficult as it requires understanding why the executions differ. Thus, again, variational traces can help understanding the causes of information flow by explaining data and control flow differences.

[1] https://github.com/meinicke/VarexJ/wiki/Coverage-PlugIn

Detecting Behavioral Feature Interactions

Variational traces are useful tool for understanding interactions in a single test case. However, if we want to analyze how all options interact in general, we need to execute multiple test cases. Thus, variational traces can become overwhelming as we cannot use our projections for subsets of options and we need to understand many variational traces, one for each test case.

Understanding many complex variational traces requires a form of abstraction to make them manageable. We developed an analysis of feature interaction in the variational traces that we present as *feature interactions graphs* [Soares et al., 2018]. These graphs summarize interactions between pairs of options including information on which variables are affected by the interaction. Beyond identifying which options interact with each other, the abstraction to feature interaction graphs enables us to perform analyses of how options interact. For example, we can detect suspicions interactions, such as suppressions, where one options disables the effect of another option [Soares et al., 2018].

5.7 Related Work

We already discussed closely related work in the domains of automatic debugging [Abreu et al., 2006, Apiwattanapong et al., 2007, Groce et al., 2006, Jones et al., 2002, Parnin and Orso, 2011, Sumner and Zhang, 2009, 2013, Xin et al., 2008, Zeller, 2002, Zeller and Hildebrandt, 2002] and feature interactions [Cohen et al., 2007, Kim et al., 2011, 2013, Lillack et al., 2017, Medeiros et al., 2016, Meinicke et al., 2016, Nguyen et al., 2014b, Nie and Leung, 2011, Souto et al., 2017, von Rhein et al., 2011] in Section 5.1. Our work combines ideas from both fields and builds specifically on the idea of sharing and coordinating multiple executions with variational execution [Meinicke, 2014, Meinicke et al., 2016, Nguyen et al., 2014b, Wong et al., 2018b] and thus sidesteps the challenges of trace alignment and full trace recording [Burtscher et al., 2005, Kanev and Cohn, 2011] as discussed.

Omniscient or *back-in-time debugging* allows exploring and debug a single execution [Burg et al., 2013, Ko and Myers, 2008, 2010, Lewis, 2003, Pothier et al., 2007]. In contrast to standard debuggers, back-in-time debuggers allow exploring information of previous parts of the execution. To do so, they record full traces resulting in severe scalability challenges as they cannot predict which information is of interest. In contrast, we provide a dynamic analysis that can decide on the fly which information is potentially relevant to explain the fault.

Symbolic execution allows to explore all executions of a program for different inputs. As discussed in Section 5.2.3, symbolic execution usually does not share the executions after they are separated unless they incorporate ideas from variational execution [Sen et al., 2015]. The *interactive verification debugger* is a tool to understand the symbolic execution of a program [Hentschel et al., 2016a,b]. The tool visualizes the execution in a tree structure similar to Varviz. However, as the symbolic execution never joins the tree structure gets large even for small programs [Kanyshkova, 2017]. In contrast, our variational trace provides a concise representation of many executions.

Static and dynamic program slicing are techniques to reduce a program to only the statements relevant for understanding the state at a given program point [Korel and Laski, 1988, Weiser, 1981]. However, program slices are often large and cannot explain differences among multiple executions, especially execution omission bugs [Zhang et al., 2007]. Differential slicing and dual slicing [Johnson et al., 2011, Kim et al., 2015, Sumner and Zhang, 2013] compare the execution of two executions and reduces the comparison using program slicing. In contrast to slicing approaches, we aim to explain the differences among many executions and only focus on the causes and differences in the state to keep the explanations concise.

Multi execution are approaches that synchronize (typically two) concrete executions. These approaches enable analyses for information flow [Devriese and Piessens, 2010, Kim et al., 2015, Kolbitsch et al., 2012, Kwon et al., 2016], configuration faults [Su et al., 2007] and inconsistent updates [Hosek and Cadar, 2013, Maurer and Brumley, 2012, Tucek et al., 2009]. In contrast, our approach can compare a potentially exponential number of executions and helps to understand how the differences affect the program behavior.

5.8 Conclusion

In this work, we propose variational traces to explain the runtime behaviors of inputs and interactions among them. Efficient generation of variational traces is only possible due to optimal alignment using variational execution [Meinicke, 2014, Wong et al., 2018b] (see Chapter 3), which enables us to generate traces for larger applications, and the properties of how options interact, such as local and orthogonal [Meinicke et al., 2016] (see Chapter 4), which are essential for variational traces to be concise. To visualize variational traces, we provide an interactive Eclipse plugin called Varviz, which enables programmers to use variational traces for debugging interaction faults. In our user study, we show that users of Varviz outperform the users of the Eclipse debugger significantly in terms of understanding and time spent on debugging tasks. Users of Varviz can focus on relevant parts of the programs quickly, without being distracted by irrelevant data and control flow decisions. When compared with users who use the standard Eclipse debugger, Varviz users can finish all the debugging and understanding tasks, using less than half of the time. We further evaluate the size of variational traces on six highly configurable systems. In general, the size of variational traces can get large, but our filters are effective in reducing the traces to a relatively small number of statements. Overall, our evaluation of effectiveness and scalability demonstrates that variational traces are useful in practice to understand differences among executions.

6. Conclusion

In this chapter, we summarize the contributions of this thesis and discuss future research directions.

Variability in software is challenging as it hinders analysis and code comprehension [Apel et al., 2013a, Melo et al., 2017, Thüm et al., 2014]. Especially, interactions among options are hard to detect, understand, and resolve [Abal et al., 2018, Apel et al., 2013b, Calder et al., 2003a, Garvin and Cohen, 2011]. With the research of this thesis, we aimed to increase our understanding of such interactions. In detail, we help detecting faults caused by feature interaction using variational execution [Meinicke, 2014, Nguyen et al., 2014a, Wong et al., 2018b], we gained a better understanding of how options interact with each other [Meinicke et al., 2016], and we help programmers understanding interactions using variational traces [Meinicke et al., 2018, Soares et al., 2018].

In Chapter 3, we discussed our work on variational execution. Variational execution itself is not the main contribution of this thesis. Though, we made several engineering contributions advancing the technique (see Chapter 3). Instead, our contribution bases on the realization that beyond the ability of executing all configurations, variational execution is an efficient approach for aligning many of executions, sidestepping challenges of trace alignment, nondeterminism, and the memory explosion caused by recording full traces. Aligning the executions of all configurations enables the work of this thesis as it allows us to directly observe feature interactions on the control and on data-flow. Observing feature interactions is challenging without variational execution as they can usually only be observed by their effects, such as faults.

Characteristics of Interactions: There have been assumptions on feature interactions, such as that only few feature interact at a time and that most faults are caused by interactions of three features [Abal et al., 2018, Garvin and Cohen, 2011, Kuhn et al., 2004]. However, these assumptions are biased towards bug reports, as interactions are only observed by undesired behavior. Existing analyses try to scale with these assumptions about interactions. However, whether these assumptions are valid and how much we can speed up analyses in future was not well

understood. Based on variational execution, we implemented a dynamic analysis to quantify different characteristics of interactions with benchmarks and to analyze real-world programs. We found that essential configuration complexity (i.e., the computational overhead) induced by real-world interactions is usually low, making configuration-complete analyses feasible. Based on our insights, we discussed typical characteristics of interactions, which can be exploited by future approaches for analyzing configurable systems.

Our new insights will help designing better quality assurance strategies that are aware of how options interact in software. These insights enable us to explain why current analyses scale for certain feature interactions but do not for others. With our new understanding on feature interactions, we help building and designing configurable software that is easier to analyze by being aware of which types of interactions challenge analysis or are supported by existing analyses.

Variational Debugging: Detecting interaction faults is only the first step. The fault also needs to be resolved, which requires understanding why and how feature interactions causes the faulty behavior. Comparing valid and failing executions has been shown the be useful for explaining faults [Sumner and Zhang, 2013, Zeller, 2002]. When understanding interactions, however, it is not sufficient to align two executions. Instead, we propose to align the executions of all configurations, using variational execution. We summarize the alignment of the executions in variational traces to explain the runtime behaviors of inputs and interactions among them. We provide an interactive Eclipse plugin called Varviz, which enables programmers to use variational traces for debugging interaction faults. We showed that users of Varviz outperform the users of the Eclipse debugger significantly in terms of understanding and time spent on debugging tasks. Users of Varviz can focus on relevant parts of the programs quickly, without being distracted by irrelevant data and control flow decisions. In general, the size of variational traces can get large, but our filters are effective in reducing the traces to a relatively small number of statements. Overall, our evaluation of effectiveness and scalability demonstrates that variational traces are useful in practice to understand differences among executions.

With variational traces, we provide the first opportunity to directly observe feature interactions when they occur during runtime. The ability to observe feature interactions directly enables future researchers to study them in more detail. Variational traces further help developers to debug interaction faults, which are notoriously hard to understand with standard tools. Beyond the usefulness of variational traces, our study shows for the first time, that comparative approaches (i.e., contrasting executions) actually help developers understanding and debugging faults.

In summary, we extended the general understanding of feature interactions. We enable efficient future analysis of configurable systems by informing them about properties of interactions. We further enable researchers to study and observe feature interactions directly using variational traces. Overall, our work helps detecting interaction faults, which improves software quality and reduce development effort. This directly impacts safety, security, reliability and cost of software.

6.1 Suggestions for Future Work

In this Section, we want to discuss future research opportunities according to variational traces and the two main parts of this thesis, variational execution, characteristics of feature interactions, and variational traces.

Variational Execution: Variational execution is a new and evolving technique that gained major improvements over the last years [Meinicke, 2014, Wong et al., 2018b]. However, there are still major challenges that need to be solved for variational execution to be usable in industrial settings. The first challenge is the environment barrier where code is execution that cannot be executed variational, such as native methods. This problem is shared with related techniques, such as symbolic execution [Baldoni et al., 2018]. A simple solution is to invoke the code multiple times, which however may cause incorrect behavior if the method has side effects (e.g., writing to a file). Thus, the environment is often modeled separately to avoid these side effects [Baldoni et al., 2018, Cadar et al., 2008a]. Such model classes, however, are often implemented manually with high effort. Automated solutions simulating the environment would have a great impact for the applicability of variational execution.

A second bottleneck of variational execution are variations in data structures. Interactions on data structures can in the worst case cause an exponential explosion, making variational execution impractical [Walkingshaw et al., 2014]. Research has shown that variational data structures can efficiently handle interactions among many options [Lazarek, 2017, Wong et al., 2018b]. However, manually lifting each data structure to be variational including specialized handling of them is again impractical. Automated and general solutions for variational data structures are needed to speed up variational execution and to avoid the exponential explosion.

Variational execution is an efficient technique to explore many variations of a system and its inputs. Beyond testing configurable systems and observing feature interactions, variational execution has more interesting applications [Wong et al., 2018a]. Future research should explore how to improve existing approaches that rely on executing a system multiple times with minor variations. Some promising applications are [Wong et al., 2018a]: (1) mutation testing, where variational execution can be used to explore many mutations at the same time including combinations of them, called higher oder mutations [Chen, 2018, Jia and Harman, 2009, 2011], and (2) automatic program repair, where minor changes are applied to a faulty program and the test suite is invoked repeatedly for each change until all tests pass [Le Goues et al., 2012].

Characteristics of Feature Interactions: We observed characteristics of interactions that are more nuanced than ”low interaction degree”. These characteristics should be considered when designing analysis of configurable systems and interactions. Especially, the insights that not all options interact and that most parts of the execution are not affected by interactions help when designing efficient approaches.

Our study on characteristics was done on a small set of systems. Future work should investigate whether our findings hold for larger systems as well and whether there are more interesting characteristics.

Variational Debugging: With variational traces we developed an approach that allows to observe feature interactions directly. We already developed an analysis based on variational traces to detect unintended behavioral interactions [Soares et al., 2018]. We believe that variational traces are helpful for developers and researchers to understand how options are interacting. However, currently variational traces can become large for long executions due to repeated and unimportant interactions. For variational traces to be a useful debugging tool even for larger systems, we need a way to reduce the traces to the essential statements necessary for understanding of an interaction (e.g., a fault). This can be done with impact analysis to only include statements that are necessary to cause a certain fault. Further heuristics, such as loop summaries, can be applied to reduce the amount of redundant information in the trace.

A. Appendix

In the context of this work we performed a user study to evaluate variational traces. In Section 5.4, we used variational traces generated by Varviz to compare their usefulness to a standard eclipse debugger when understanding faults caused by variations. In this Section, we present the *complete* variational traces that we showed the users in our study to illustrate why variational traces are helpful.

A.1 Variational Trace for Elevator

In Figure A.1, we show the variational trace for Elevator used in our user study of Section 5.4. As shown, the exception is thrown in the method `checkAfterTimeShift`. The method checks whether the `expectedDirection` is equivalent to the actual direction (`getCurrentDirection`) of the elevator. The trace shows the values of both: `expectedDirection` is `DOWN` in all configurations and the current direction is `UP` if *executiveFloor* is selected, `DOWN` otherwise. As the directions are not equal if *executiveFloor* is selected, the Exception is thrown under the condition of *executiveFloor*.

The next step is to find the place where the direction is changed. This leads us to the method `continueInDirection`. The method is invoked with different directions depending on the selection of *executiveFloor*. This direction change is caused by the previous check of `stopRequestedInDirection`, which returns false if *executiveFloor* is selected. The implementation of *executiveFloor* overrides any other call to force the elevator to go to the executed floor.

The connections among these method calls are difficult to detect as they do not appear in the stack trace of the exception. A further challenge when using a debugger is that the method `timeShift` is invoked multiple times before the exception is thrown which makes debugging using breakpoints difficult. As discussed in Section 5.4, we can see from the trace that the users are focus on five methods, while they can observe the states and the state differences that lead to the fault.

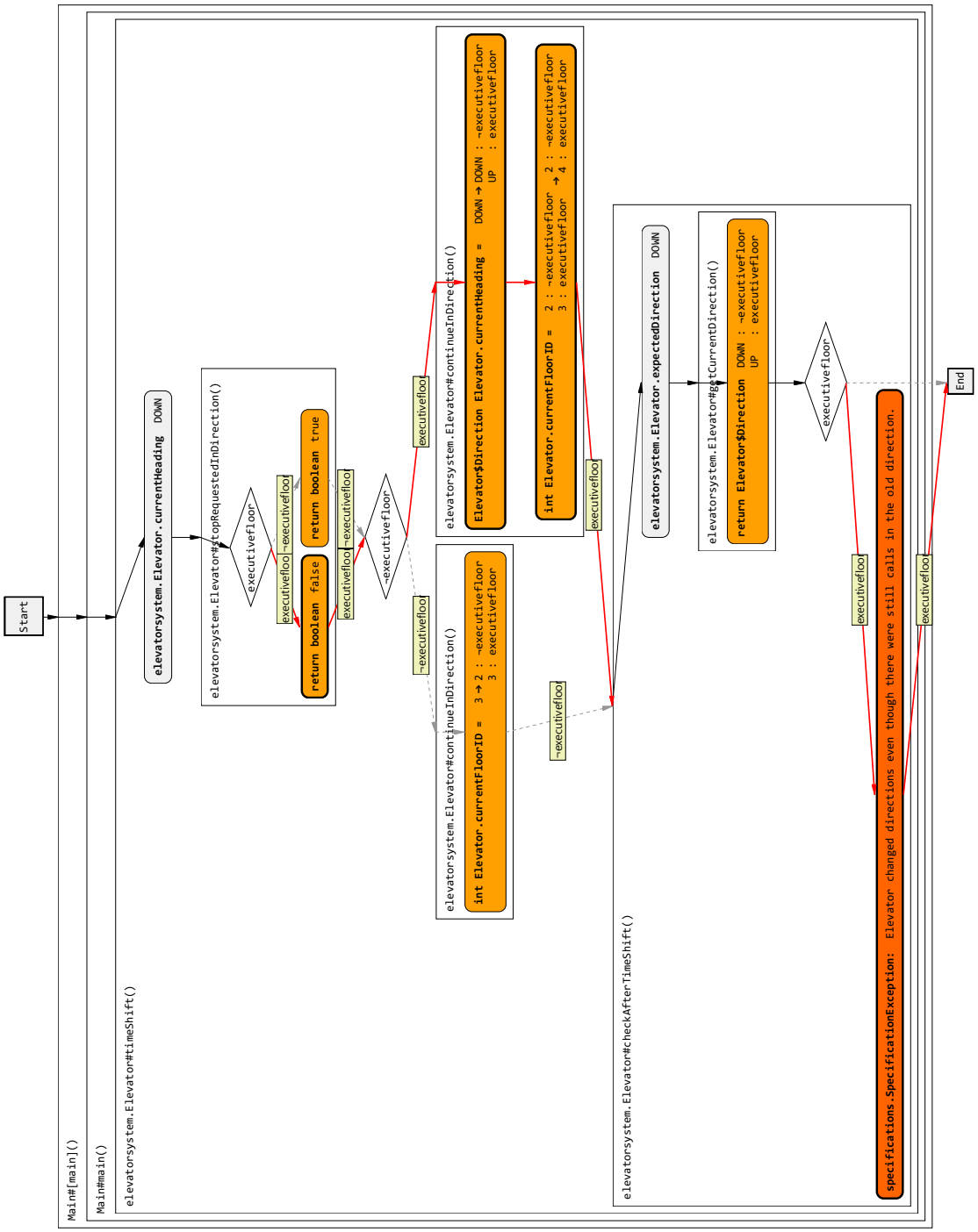


Figure A.1: Complete variational trace for Elevator used in our user study (Section 5.4).

A.2 Variational Trace for NanoXML

In Figure A.2, we show the complete variational trace for NanoXML used in our user study of Section 5.4. The trace illustrates the main challenge of unimportant statements and state differences that are not important to understand the fault. Most of the statements in the trace show minor differences in characters. These differences are caused by reading two different files. Thus, when parsing the strings

simultaneously, the elements do not align, and the variational trace reports a difference.

Even though the trace contains a lot of noise, the participants were able to detect the important statements in the trace easily. The exception is thrown because the `str` does not equal `prefix + name` (see Listing A.1). When looking in the trace, we can see all the values used in the if statement. We can see that `prefix + name` under condition *testva_ns* is `"ns:" + ":Bar"`. Thus, there are two colons instead of one as expected.

To solve the fault, we need to figure out why there is an additional colon. When going back on the trace, we can find the location where `prefix` and `name` are changed under condition *testva_ns*. These changes are automatically highlighted by Varviz due to the red arrows. Opening the location of the statements leads us to the code shown in Listing A.2 (this is supported by double clicking on the statement in Varviz). We can see that the method `substring` on `name` is called with the parameter `colonIndex`. As substring is inclusive, `name` contains the additional colon. Thus, changing the parameter to `colonIndex+1` will fix the bug.

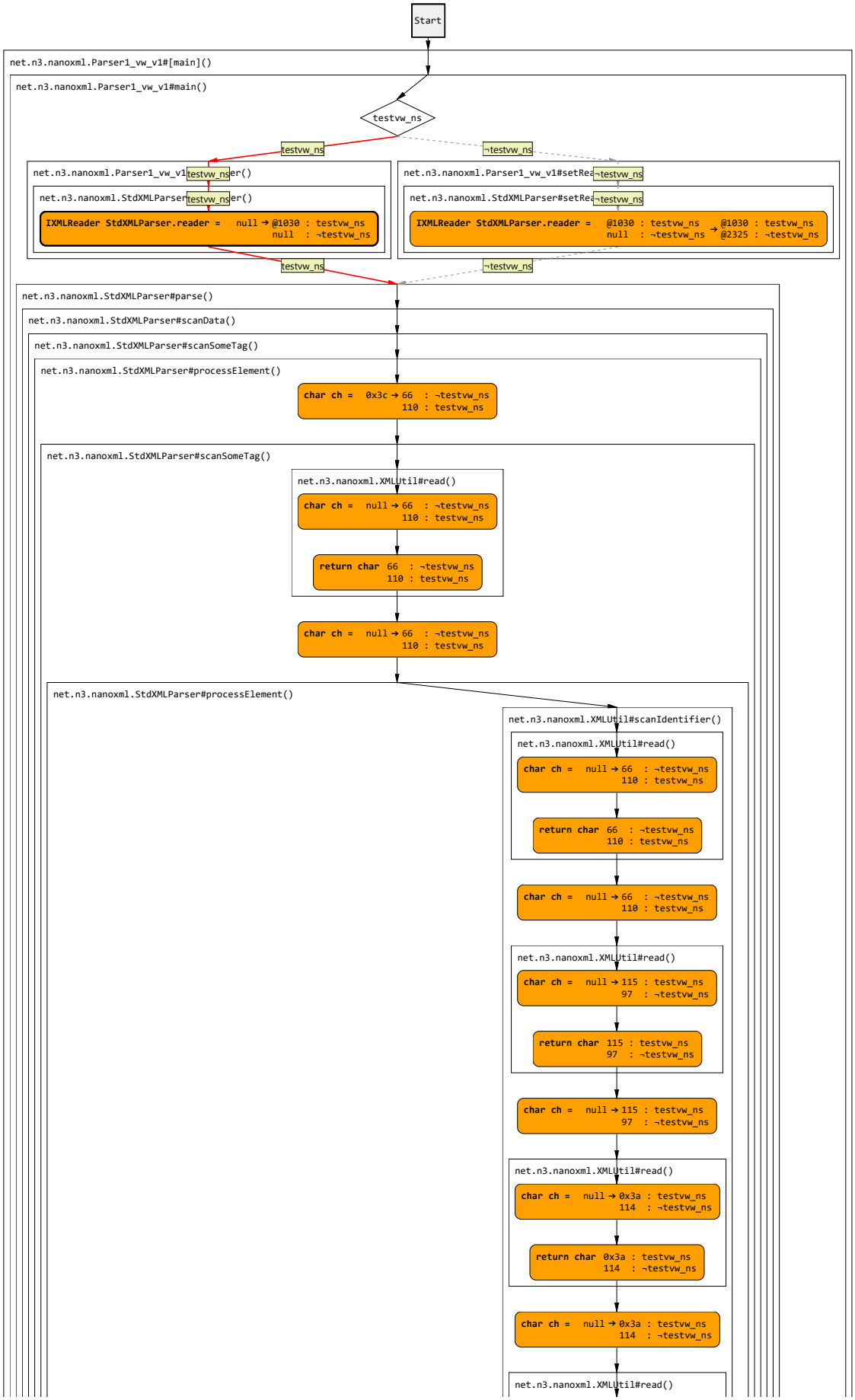
In summary, debugging the fault using a standard debugger is again challenging as the method `processElement` is called recursively many times before the exception is thrown. Varviz instead highlights the exact location and state causing the fault. Thus, even though the variational trace contains unnecessary statements, it is eases debugging.

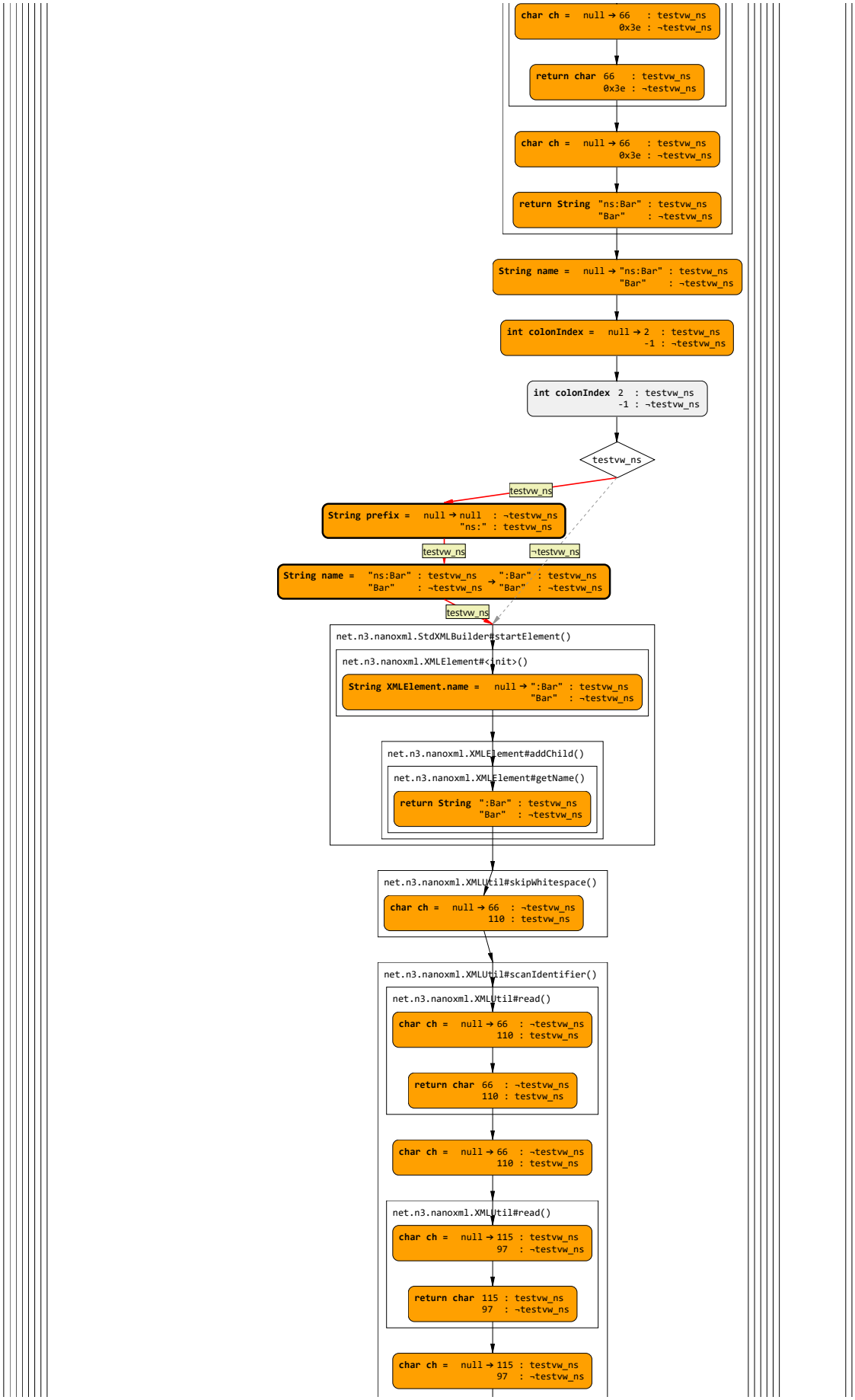
```
1 if (!str.equals(prefix==null?name:prefix+name))
2   XMLUtil.errorWrongClosingTag(this.reader, name, str);
```

Listing A.1: Parser check in NanoXML.

```
1 String name = XMLUtil.scanIdentifier();
2 String prefix = null;
3 int colonIndex = name.indexOf(':');
4 if (colonIndex > 0) {
5   prefix = name.substring(0, colonIndex+1);
6   name = name.substring(colonIndex);
7 }
```

Listing A.2: Location of the fault in NanoXML.


Start
net.n3.nanoxml.Parser1_vw_v1#[main]()
net.n3.nanoxml.Parser1_vw_v1#main()
testvw_ns
testvw_ns
¬testvw_ns
net.n3.nanoxml.Parser1_vw_v1
testvw_ns
er()
net.n3.nanoxml.StdXMLParser
testvw_ns
er()
IXMLReader StdXMLParser.reader = null → @1030 : testvw_ns
null : ¬testvw_ns
net.n3.nanoxml.Parser1_vw_v1#setRea
¬testvw_ns
net.n3.nanoxml.StdXMLParser#setRea
¬testvw_ns
IXMLReader StdXMLParser.reader = @1030 : testvw_ns → @1030 : testvw_ns
null : ¬testvw_ns @2325 : ¬testvw_ns
testvw_ns
¬testvw_ns
net.n3.nanoxml.StdXMLParser#parse()
net.n3.nanoxml.StdXMLParser#scanData()
net.n3.nanoxml.StdXMLParser#scanSomeTag()
net.n3.nanoxml.StdXMLParser#processElement()
char ch = 0x3c → 66 : ¬testvw_ns
110 : testvw_ns
net.n3.nanoxml.StdXMLParser#scanSomeTag()
net.n3.nanoxml.XMLUtil#read()
char ch = null → 66 : ¬testvw_ns
110 : testvw_ns
return char 66 : ¬testvw_ns
110 : testvw_ns
char ch = null → 66 : ¬testvw_ns
110 : testvw_ns
net.n3.nanoxml.StdXMLParser#processElement()
net.n3.nanoxml.XMLUtil#scanIdentifier()
net.n3.nanoxml.XMLUtil#read()
char ch = null → 66 : ¬testvw_ns
110 : testvw_ns
return char 66 : ¬testvw_ns
110 : testvw_ns
char ch = null → 66 : ¬testvw_ns
110 : testvw_ns
net.n3.nanoxml.XMLUtil#read()
char ch = null → 115 : testvw_ns
97 : ¬testvw_ns
return char 115 : testvw_ns
97 : ¬testvw_ns
char ch = null → 115 : testvw_ns
97 : ¬testvw_ns
net.n3.nanoxml.XMLUtil#read()
char ch = null → 0x3a : testvw_ns
114 : ¬testvw_ns
return char 0x3a : testvw_ns
114 : ¬testvw_ns
char ch = null → 0x3a : testvw_ns
114 : ¬testvw_ns
net.n3.nanoxml.XMLUtil#read()


char ch = null → 66 : testvw_ns
0x3e : ¬testvw_ns
return char 66 : testvw_ns
0x3e : ¬testvw_ns
char ch = null → 66 : testvw_ns
0x3e : ¬testvw_ns
return String "ns:Bar" : testvw_ns
"Bar" : ¬testvw_ns
String name = null → "ns:Bar" : testvw_ns
"Bar" : ¬testvw_ns
int colonIndex = null → 2 : testvw_ns
-1 : ¬testvw_ns
int colonIndex 2 : testvw_ns
-1 : ¬testvw_ns
testvw_ns
testvw_ns
String prefix = null → null : ¬testvw_ns
"ns:" : testvw_ns
testvw_ns
¬testvw_ns
String name = "ns:Bar" : testvw_ns
"Bar" : ¬testvw_ns → ":Bar" : testvw_ns
"Bar" : ¬testvw_ns
testvw_ns
net.n3.nanoxml.StdXMLBuilder#startElement()
net.n3.nanoxml.XMLElement#<init>()
String XMLElement.name = null → ":Bar" : testvw_ns
"Bar" : ¬testvw_ns
net.n3.nanoxml.XMLElement#addChild()
net.n3.nanoxml.XMLElement#getName()
return String ":Bar" : testvw_ns
"Bar" : ¬testvw_ns
net.n3.nanoxml.XMLUtil#skipWhitespace()
char ch = null → 66 : ¬testvw_ns
110 : testvw_ns
net.n3.nanoxml.XMLUtil#scanIdentifier()
net.n3.nanoxml.XMLUtil#read()
char ch = null → 66 : ¬testvw_ns
110 : testvw_ns
return char 66 : ¬testvw_ns
110 : testvw_ns
char ch = null → 66 : ¬testvw_ns
110 : testvw_ns
net.n3.nanoxml.XMLUtil#read()
char ch = null → 115 : testvw_ns
97 : ¬testvw_ns
return char 115 : testvw_ns
97 : ¬testvw_ns
char ch = null → 115 : testvw_ns
97 : ¬testvw_ns

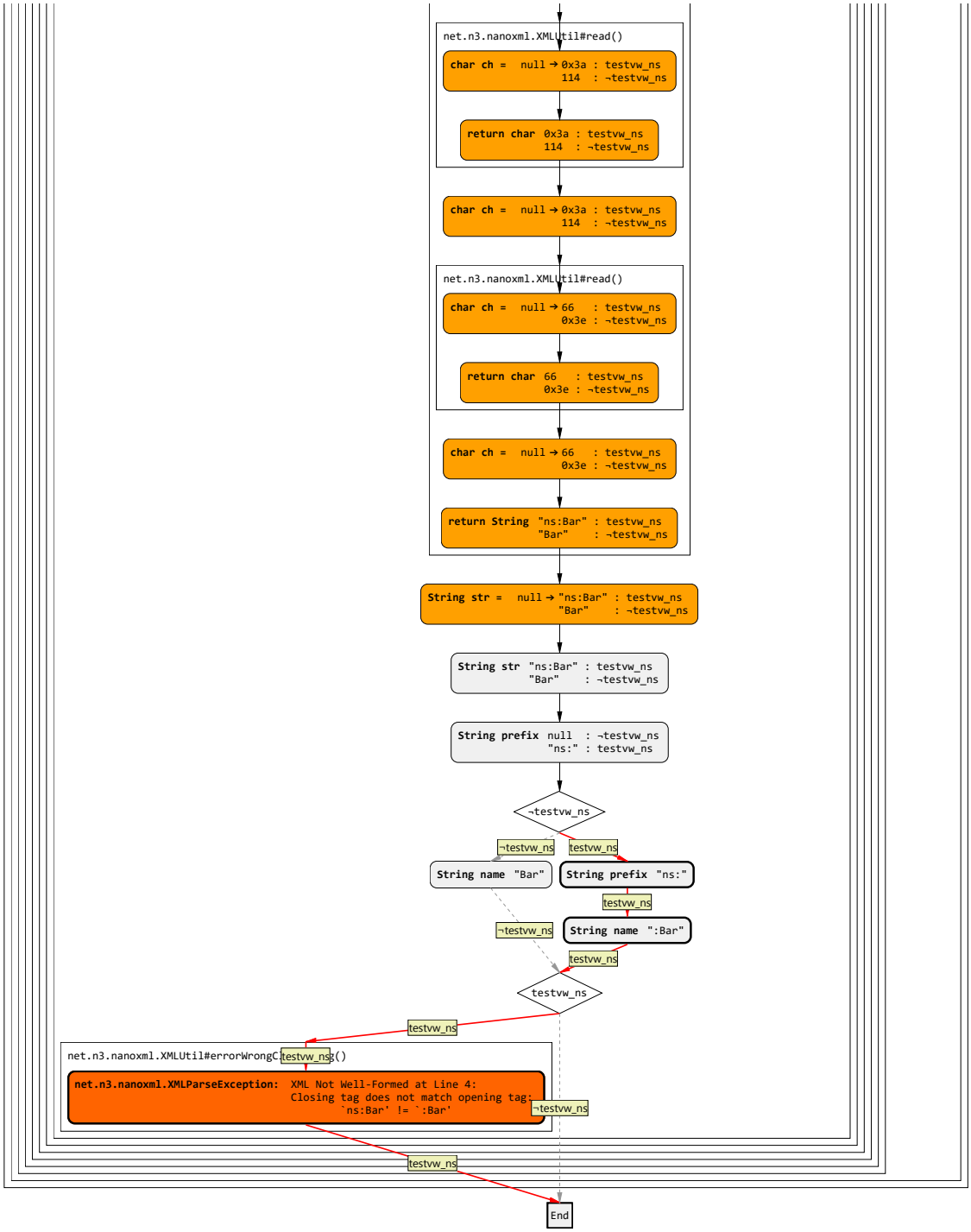


Figure A.2: Complete variational trace for NanoXML used in our user study (Section 5.4).

Bibliography

Iago Abal, Jean Melo, Ştefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):10, 2018. (cited on Page 1, 2, 3, 7, 8, 9, 10, 27, 31, 39, 42, 44, 47, 57, 69, 71, and 75)

Rui Abreu, Peter Zoeteweij, and Arjan J C Van Gemund. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 39–46. IEEE, 2006. (cited on Page 49 and 72)

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 International Conference on Generative Programming and Component Engineering (GPCE)*, pages 144–155. ACM, 2016a. (cited on Page 9)

Mustafa Al-Hajjaji, Jens Meinicke, Sebastian Krieter, Reimar Schröter, Thomas Thüm, Thomas Leich, and Gunter Saake. Tool Demo: Testing Configurable Systems with FeatureIDE. *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, page 173, 2016b. (cited on Page 9)

Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. Effective Product-Line Testing Using Similarity-Based Product Prioritization. *Software & Systems Modeling (SOSYM)*, pages 1–23, 2017. (cited on Page 9)

Saswt Anand, Corina S Păsăreanu, and Willem Visser. JPF-SE: A Symbolic Execution Extension to Java Pathfinder. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 134–138. Springer, 2007. (cited on Page 32, 34, and 36)

Sven Apel, Sergiy Kolesnikov, Jörg Liebig, Christian Kästner, Martin Kuhlemann, and Thomas Leich. Access Control in Feature-Oriented Programming. *Science of Computer Programming (SCP)*, 77(3):174–187, 2012. (cited on Page 11)

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. Feature-Oriented Software Product Lines: Concepts and Implementation. Springer, 2013a. (cited on Page 1, 7, 51, and 75)

Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge. In *Proceedings of the International SPLC Workshop Feature-Oriented Software Development (FOSD)*, pages 1–8, 2013b. ACM. (cited on Page 1, 7, 47, and 75)

Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 482–491. IEEE, 2013c. (cited on Page 41)

Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. JDiff: A Differencing Technique and Tool for Object-Oriented Programs. *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 14(1): 3–36, 2007. (cited on Page 50 and 72)

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, 49(6):259–269, 2014. (cited on Page 51)

Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. Design and Validation of Variability in Product Lines. In *Proceedings of the International Workshop on Product Line Approaches in Software Engineering (PLEASE)*, pages 25–30, 2011. ACM. (cited on Page 28)

Thomas H Austin and Cormac Flanagan. Multiple Facets for Dynamic Information Flow. *ACM Sigplan Notices*, 47(1):165–178, 2012a. (cited on Page 17 and 50)

Thomas H. Austin and Cormac Flanagan. Multiple Facets for Dynamic Information Flow. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 165–178. ACM, 2012b. (cited on Page 27 and 71)

Thomas H Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. Faceted Execution of Policy-Agnostic Programs. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 15–26. ACM, 2013. (cited on Page 2, 27, and 71)

Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys (CSUR)*, 51(3):50:1–50:39, 2018. (cited on Page 10, 26, 28, 37, 45, 57, and 77)

Thomas Ball, Vladimir Levin, and Sriram K Rajamani. A Decade of Software Model Checking with SLAM. *Communications of the ACM*, 54(7):68–76, 2011. (cited on Page 28)

Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The Software Model Checker Blast: Applications to Software Engineering. *International*

Journal on Software Tools for Technology Transfer (STTT), 9(5):505–525, 2007. (cited on Page 28)

Hugh Beyer and Karen Holtzblatt. Contextual Design: Defining Customer-Centered Systems. Elsevier, 1997. (cited on Page 62)

Glenn Bruns. Foundations for Features. In *Feature Interactions in Telecommunications and Software Systems VIII*, pages 3–11. IOS Press, 2005. (cited on Page 1 and 7)

Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Symposium on Logic in Computer Science (LICS)*, pages 428–439. IEEE, 1990. (cited on Page 50)

Brian Burg, Richard Bailey, Andrew J Ko, and Michael D Ernst. Interactive Record/Replay for Web Application Debugging. In *Proceedings of ACM symposium on User interface software and technology (SIGHCI)*, pages 473–484. ACM, 2013. (cited on Page 72)

Martin Burtscher, Ilya Ganusov, Sandra J Jackson, Jian Ke, Paruj Ratanaworabhan, and Nana B Sam. The VPC Trace-Compression Algorithms. *IEEE Journal of Transactions on Computers (TC)*, 54(11):1329–1344, 2005. (cited on Page 50, 55, and 72)

Isis Cabral, Myra B Cohen, and Gregg Rothermel. Improving the Testing and Testability of Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 241–255. Springer, 2010. (cited on Page 9, 27, 44, 57, 69, and 71)

Cristian Cadar, Daniel Dunbar, and Dawson R Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008a. (cited on Page 77)

Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008b. (cited on Page 26)

Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1066–1071. ACM, 2011. (cited on Page 57)

Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41 (1):115–141, 2003a. (cited on Page 1, 7, 47, and 75)

Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41 (1):115–141, 2003b. (cited on Page 1 and 7)

Ivan Do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *International Journal on Information and Software Technology(IST)*, 56(10):1183–1199, 2014. (cited on Page 1 and 8)

Gary Charness, Uri Gneezy, and Michael A Kuhn. Experimental Methods: Between-Subject and Within-Subject Design. *Journal of Economic Behavior & Organization*, 81(1):1–8, 2012. (cited on Page 62 and 67)

Serena Chen. Finding Higher Order Mutants Using Variational Execution. Technical Report 1809.04563, arXiv, 2018. Accepted to SPLASH'18 Student Research Competition. (cited on Page 77)

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. Model Checking. 1999. (cited on Page 10)

Lori A Clarke. A Program Testing System. In *Proceedings of the 1976 Annual Conference*, pages 488–491. ACM, 1976. (cited on Page 10, 15, 28, and 56)

Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 335–344, 2010. ACM. (cited on Page 28)

Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic Model Checking of Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 321–330, 2011. ACM. (cited on Page 28 and 50)

Paul Clements and Linda Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley, 2001. (cited on Page 1 and 7)

Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 129–139. ACM, 2007. (cited on Page 7, 9, 27, 44, 47, 50, 57, 69, 71, and 72)

Rui Gustavo Crespo, Miguel Carvalho, and Luigi Logrippo. Distributed Resolution of Feature Interactions for Internet Applications. *Computer Networks*, 51(2):382–397, 2007. (cited on Page 1)

Marcelo d'Amorim, Steven Lauterburg, and Darko Marinov. Delta Execution for Efficient State-Space Exploration of Object-Oriented Programs. *IEEE Transactions on Software Engineering (TSE)*, 34(5):597–613, 2008. (cited on Page 21)

Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. FlowFox: A Web Browser with Flexible and Precise Information Flow Control. In *Proceedings of the International Conference on Conference on Computer and Communications Security (CCS)*, pages 748–759. ACM, 2012. (cited on Page 28)

Dominique Devriese and Frank Piessens. Noninterference Through Secure Multi-Execution. In *Symposium on Security and Privacy (SP)*, pages 109–124. IEEE, 2010. (cited on Page 28, 71, and 73)

Robin Donaldson and Muffy Calder. Modular Modelling of Signalling Pathways and Their Cross-Talk. *Theoretical Computer Science (TCS)*, 456:30–50, 2012. (cited on Page 1)

Faezeh Ensan, Ebrahim Bagheri, and Dragan Gašević. Evolutionary Search-based Test Generation for Software Product Line Feature Models. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 613–628. Springer, 2012. (cited on Page 9)

Martin Erwig and Eric Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(1):6:1–6:27, 2011a. (cited on Page 16, 19, 28, and 56)

Martin Erwig and Eric Walkingshaw. Variation Programming with the Choice Calculus. In *Generative and Transformational Techniques in Software Engineering IV*, pages 55–100. Springer, 2011b. (cited on Page 16)

Martin Erwig and Eric Walkingshaw. Variation Programming with the Choice Calculus. In *Proceedings of the summer school on Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 55–100, 2013. Springer. (cited on Page 16 and 19)

Janet Feigenspan, Michael Schulze, Maria Papendieck, Christian Kästner, Raimund Dachselt, Veit Köppen, and Mathias Frisch. Using Background Colors to Support Program Comprehension in Software Product Lines. In *Proceedings of the Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 66–75. IEEE, 2011. (cited on Page 51)

Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. Do Background Colors Improve Program Comprehension in the #Ifdef Hell? *Empirical Software Engineering (EMSE)*, 18(4):699–745, 2013. (cited on Page 51 and 71)

Brady J Garvin and Myra B Cohen. Feature Interaction Faults Revisited: An Exploratory Study. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 90–99. IEEE, 2011. (cited on Page 1, 2, 7, 8, 9, 27, 31, 39, 42, 45, 47, 57, 69, 71, and 75)

Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. In *Proceedings of the International Conference on Conference on Computer and Communications Security (CCS)*, pages 38–49. ACM, 2012. (cited on Page 1)

Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Test Confessions: A Study of Testing Practices for Plug-in Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 244–254, 2012. IEEE. (cited on Page 1 and 8)

Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error Explanation with Distance Metrics. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(3):229–247, 2006. (cited on Page 50 and 72)

Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. Test Them All, is it Worth it? A Ground Truth Comparison of Configuration Sampling Strategies. *arXiv preprint arXiv:1710.07980*, 2017. (cited on Page 1, 8, and 10)

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 (EMSE)*, pages 1–44, 2018. (cited on Page 2 and 45)

Robert J. Hall. Fundamental Nonmodularity in Electronic Mail. *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 12(1): 41–79, 2005. (cited on Page 39 and 45)

Klaus Havelund and Thomas Pressburger. Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000. (cited on Page 24, 32, and 34)

Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutation-based generation of software product line test configurations. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE)*, pages 92–106. Springer, 2014. (cited on Page 9)

M Hentschel, R Hähnle, and R Bubel. The Interactive Verification Debugger: Effective Understanding of Interactive Proof Attempts. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 846–851, 2016a. (cited on Page 72)

Martin Hentschel, Reiner Hähnle, and Richard Bubel. An Empirical Evaluation of Two User Interfaces of an Interactive Program Verifier. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 403–413, 2016b. (cited on Page 57 and 72)

Petr Hosek and Cristian Cadar. Safe Software Updates via Multi-Version Execution. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 612–621. IEEE, 2013. (cited on Page 28 and 73)

Petr Hosek and Cristian Cadar. VARAN the Unbelievable: An Efficient N-version Execution Framework. In *ASPLOS*, pages 339–353, 2015. (cited on Page 28)

Praveen Jayaraman, Jon Whittle, Ahmed M Elkhodary, and Hassan Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical

Pair Analysis. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 151–165. Springer, 2007. (cited on Page 1)

Yue Jia and Mark Harman. Higher Order Mutation Testing. *International Journal on Information and Software Technology(IST)*, 51(10):1379–1393, 2009. (cited on Page 77)

Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering (TSE)*, 37(5):649–678, 2011. (cited on Page 77)

Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 46–55, 2012. ACM. (cited on Page 9)

Noah M Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *Symposium on Security and Privacy (SP)*, pages 347–362. IEEE, 2011. (cited on Page 49, 50, and 73)

James A Jones, Mary Jean Harrold, and John Stasko. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 467–477. ACM, 2002. (cited on Page 49, 50, and 72)

Alma L Juarez Dominguez. Detection of Feature Interactions in Automotive Active Safety Features. PhD thesis, University of Waterloo, 2012. (cited on Page 1)

Svilen Kanev and Robert Cohn. Portable trace compression through instruction interpretation. In *Proceedings of of the International IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 107–116. IEEE, 2011. (cited on Page 50, 55, and 72)

Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. (cited on Page 9 and 57)

Maria Kanyshkova. Symbolisches Debuggen von Software-Produktlinien. Bachelor's thesis, University of Magdeburg, Germany, 2017. (cited on Page 72)

Christian Kästner. Differential Testing for Variational Analyses: Experience from Developing KConfigReader. *arXiv arXiv:1706.09357*, 2017. (cited on Page 24)

Christian Kästner and Sven Apel. Virtual Separation of Concerns - A Second Chance for Preprocessors. *Journal of Object Technology (JOT)*, 8(6):59–78, 2009. (cited on Page 51)

Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A Variability-Aware Module System. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 773–792, 2012a. ACM. (cited on Page 11 and 45)

Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward Variability-Aware Testing. In *Proceedings of the International SPLC Workshop Feature-Oriented Software Development (FOSD)*, pages 1–8, 2012b. ACM. (cited on Page 2, 19, 23, and 27)

Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. Eliminating Products to Test in a Software Product Line. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 139–142, 2010. ACM. (cited on Page 50)

Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. Reducing Combinatorics in Testing Product Lines. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 57—68, 2011. ACM. (cited on Page 10, 43, 51, and 72)

Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. Shared Execution for Efficiently Testing Product Lines. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 221–230, 2012. IEEE. (cited on Page 11, 27, 41, and 50)

Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 257–267. ACM, 2013. (cited on Page 10, 32, 34, 37, 43, 50, and 72)

Dohyeong Kim, Yonghwi Kwon, William N. Sumner, Xiangyu Zhang, and Dongyan Xu. Dual Execution for On the Fly Fine Grained Execution Comparison. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 325–338, 2015. (cited on Page 50, 55, and 73)

James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976. (cited on Page 10, 15, 28, and 56)

Andrew J. Ko and Brad A. Myers. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In *Proceedings of the International Conference on Software Engineering (ICSE)*, page 301, 2008. (cited on Page 50, 51, and 72)

Andrew J. Ko and Brad A. Myers. Extracting and Answering Why and Why Not Questions About Java Program Output. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(2):4:1–4:36, 2010. (cited on Page 48 and 72)

Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An Exploratory Study of How Eevelopers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks. *IEEE Transactions on Software Engineering (TSE)*, 32(12), 2006a. (cited on Page 57 and 66)

Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering (TSE)*, 32(12):971–987, 2006b. (cited on Page 51 and 71)

M. Kolberg, E.H. Magill, D. Marples, and S. Reiff. Feature Interactions in Telecommunication Systems VI, chapter Results of the Second Feature Interaction Contest, pages 311–325. IOS Press, 2000. (cited on Page 45)

Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-Cloaking Internet Malware. In *IEEE Symposium on Security and Privacy (SP)*, pages 443–457. IEEE, 2012. (cited on Page 28, 71, and 73)

Bogdan Korel and Janusz Laski. Dynamic Program Slicing. *Information processing letters*, 29(3):155–163, 1988. (cited on Page 73)

Jeff Kramer, Jeff Magee, Morris Sloman, and Andrew Lister. CONIC: An Integrated Approach to Distributed Computer Control Systems. *IEE Proceedings E (Computers and Digital Techniques)*, 130(1):1–10, 1983. (cited on Page 39)

D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering (TSE)*, 30:418–421, 2004. (cited on Page 7, 8, 9, 27, 31, 39, 42, 45, 57, 69, 71, and 75)

Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient State Merging in Symbolic Execution. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, number EPFL-CONF-176487, pages 193–204. ACM, 2012. (cited on Page 28)

Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. LDX: Causality Inference by Lightweight Dual Execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 503–515. ACM, 2016. (cited on Page 2, 50, 55, 56, 71, and 73)

H. Larsson, E. Lindqvist, and R. Torkar. Outliers and Replication in Software Engineering. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 207–214. IEEE, 2014. (cited on Page 68)

Thomas D LaToza, David Garlan, James D Herbsleb, and Brad A Myers. Program Comprehension as Fact Finding. In *Proceedings of the International Symposium Foundations of Software Engineering (FSE)*, pages 361–370. ACM, 2007. (cited on Page 65 and 71)

Kim Lauenroth, Klaus Pohl, and Simon Toehning. Model Checking of Domain Artifacts in Product Line Engineering. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 269–280, 2009. IEEE. (cited on Page 28)

Lukas Lazarek. How to Efficiently Process 100 List Variations. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 36–38. ACM, 2017. (cited on Page 24, 25, 26, 29, and 77)

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering (TSE)*, 38(1):54, 2012. (cited on Page 77)

Bil Lewis. Debugging Backwards in Time. *Computing Research Repository (CoRR)*, 2003. (cited on Page 72)

Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable Analysis of Variable Software. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 81–91, 2013. ACM. (cited on Page 9)

Max Lillack, Christian Kästner, and Eric Bodden. Tracking Load-Time Configuration Options. *IEEE Transactions on Software Engineering (TSE)*, 2017. (cited on Page 2, 3, 51, 55, 71, and 72)

Roberto E. Lopez-Herrejon and Don Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *Proceedings of the International Conference on Generative and Component-Based Software Engineering (GCSE)*, pages 10–24, 2001. Springer. (cited on Page 39)

Matthew Maurer and David Brumley. Tachyon: Tandem Execution for Efficient Live Patch Testing. In *USENIX Security Symposium*, pages 617–630, 2012. (cited on Page 28 and 73)

Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 495–518, 2015. Schloss Dagstuhl–LZI. (cited on Page 1, 8, 9, and 47)

Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 664–675. ACM, 2016. (cited on Page 7, 9, 10, 27, 45, 47, 50, 57, 69, 71, and 72)

Jens Meinicke. JML-Based Verification for Feature-Oriented Programming. Bachelor's thesis, University of Magdeburg, Germany, 2013. (cited on Page 10)

Jens Meinicke. VarexJ: A Variability-Aware Interpreter for Java Applications. Master's thesis, University of Magdeburg, 2014. (cited on Page 2, 4, 11, 13, 15, 18, 19, 23, 24, 26, 27, 28, 32, 46, 48, 49, 56, 58, 72, 73, 75, and 77)

Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, and Gunter Saake. An Overview on Analysis Tools for Software Product Lines. In *Proceedings of the Workshop on Software Product Line Analysis Tools (SPLat)*, pages 94–101, 2014. ACM. (cited on Page 1)

Jens Meinicke, Chu Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. On Essential Configuration Complexity : Measuring Interactions in Highly-Configurable Systems. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 483–494, 2016. (cited on Page 2, 4, 11, 13, 15, 18, 20, 24, 26, 27, 28, 31, 45, 48, 49, 50, 56, 57, 58, 69, 71, 72, 73, and 75)

Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. Mastering Software Variability with FeatureIDE. Springer, 2017. (cited on Page 51)

Jens Meinicke, Chu-Pan Wong, Christian Kästner, and Gunter Saake. Understanding Differences among Executions with Variational Traces. *arXiv e-prints, arXiv:1807.03837*, 2018. (cited on Page 15, 18, 20, 24, 28, 46, 47, 71, and 75)

Jean Melo, Claus Brabrand, and Andrzej Wąsowski. How Does the Degree of Variability Affect Bug Finding? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 679–690. ACM, 2016. (cited on Page 4, 7, 8, 11, 12, 21, and 60)

Jean Melo, Fabricio Batista Narcizo, Dan Witzner Hansen, Claus Brabrand, and Andrzej Wasowski. Variability through the Eyes of the Programmer. In *International Conference on Program Comprehension*. IEEE, 2017. (cited on Page 75)

Meng Meng, Jens Meinicke, Chu-Pan Wong, Eric Walkingshaw, and Christian Kästner. A Choice of Variational Stacks: Exploring Variational Data Structures. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 28–35. ACM, 2017. (cited on Page 15, 24, 25, and 29)

Saul B Needleman and Christian D Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of molecular biology*, 48(3):443–453, 1970. (cited on Page 55)

Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Building Call Graphs for Embedded Client-Side Code in Dynamic Web Applications. In *Proceedings of the International Symposium Foundations of Software Engineering (FSE)*, pages 518–529, 2014a. ACM. (cited on Page 23 and 75)

Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 907–918, 2014b. ACM. (cited on Page 2, 9, 11, 15, 19, 23, 27, 28, 31, 32, 45, 48, 50, 56, and 72)

ThanhVu Nguyen, Ugur Koc, Javran Cheng, Jeffrey S. Foster, and Adam A. Porter. iGen: Dynamic Interaction Inference for Configurable Software. *Proceedings of*

the International Symposium Foundations of Software Engineering (FSE), pages 655–665, 2016. (cited on Page 51)

Armstrong Nhlabatsi, Robin Laney, and Bashar Nuseibeh. Feature Interaction: The Security Threat from within Software Systems. *Progress in Informatics*, pages 75–89, 2008. (cited on Page 1, 7, 8, and 9)

Changhai Nie and Hareton Leung. A Survey of Combinatorial Testing. *ACM Computing Surveys (CSUR)*, 43(2):11:1–11:29, 2011. (cited on Page 7, 8, 9, 27, 45, 47, 50, 57, 69, 71, and 72)

Jakob Nielsen. Estimating the Number of Subjects Needed for a Thinking Aloud Test. *International Journal of Human-Computer Studies(IJHCS)*, 41(3):385–397, 1994. (cited on Page 63)

Peter Ohmann and Ben Liblit. Lightweight Control-Flow Instrumentation and Postmortem Analysis in Support of Debugging. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 865–904. Springer, 2017. (cited on Page 13)

Steve R. Palmer and Mac Felsing. A Practical Guide to Feature-Driven Development. Pearson Education, 1st edition, 2001. (cited on Page 7)

Chris Parnin and Alessandro Orso. Are Automated Debugging Techniques Actually Helping Programmers? In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 199–209. ACM, 2011. (cited on Page 50, 60, 61, 62, 65, 71, and 72)

Malte Plath and Mark Ryan. Feature Integration Using a Feature Construct. *Science of Computer Programming (SCP)*, 41(1):53–84, 2001. (cited on Page 39, 40, and 61)

Klaus Pohl, Günter Böckle, and Frank J. van der Linden. Software Product Line Engineering: Foundations, Principles and Techniques. Springer, 2005. (cited on Page 1 and 7)

Guillaume Pothier and Éric Tanter. Back to the Future: Omniscient Debugging. *IEEE software*, 26(6), 2009. (cited on Page 48)

Guillaume Pothier, Éric Tanter, and José Piquer. Scalable Omniscient Debugging. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 535–552. ACM, 2007. (cited on Page 72)

Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443, 1997. Springer. (cited on Page 51)

Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S Foster, and Adam Porter. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 445–454. ACM, 2010. (cited on Page 2, 3, 10, 31, 32, 45, 51, and 71)

Márcio Ribeiro, Paulo Borba, and Christian Kästner. Feature Maintenance with Emergent Interfaces. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 989–1000, 2014. ACM. (cited on Page 11)

Bernard Rosner. Fundamentals of Biostatistics. Nelson Education, 2015. (cited on Page 63 and 67)

Julia Rubin and Marsha Chechik. N-Way Model Merging. In *Proceedings of the International Symposium Foundations of Software Engineering (FSE)*, pages 301–311. ACM, 2013. (cited on Page 55)

Johnny Saldaña. The Coding Manual for Qualitative Researchers. Sage, 2015. (cited on Page 62 and 65)

Martin Schäler, Alexander Grebhahn, Reimar Schröter, Sandro Schulze, Veit Köppen, and Gunter Saake. QuEval: Beyond High-Dimensional Indexing à la Carte. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1654–1665. VLDB Endowment, 2013. (cited on Page 39)

Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. Faceted Dynamic Information Flow via Control and Data Monads. In *Proceedings of the International Conference on Principles of Security and Trust (POST)*, pages 3–23. Springer, 2016. (cited on Page 2 and 71)

Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. Faceted Secure Multi Execution. In *Proceedings of the International Conference on Conference on Computer and Communications Security (CCS)*. ACM, 2018. (cited on Page 2)

Margrit Schreier. Qualitative Content Analysis in Practice. SAGE Publications, 2012. (cited on Page 62 and 65)

Koushik Sen, George Necula, Liang Gong, and Wontae Choi. MultiSE: Multi-Path Symbolic Execution Using Value Summaries. In *Proceedings of the International Symposium Foundations of Software Engineering (FSE)*, pages 842–853. ACM, 2015. (cited on Page 21, 28, 37, 45, 57, and 72)

Larissa Rocha Soares, Jens Meinicke, Sarah Nadi, Christian Kästner, and Eduardo Santana de Almeida. Exploring Feature Interactions Without Specifications: A Controlled Experiment. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 2018. (cited on Page 18, 68, 72, 75, and 78)

Sabrina Souto, Marcelo d'Amorim, and Rohit Gheyi. Balancing Soundness and Efficiency for Practical Testing of Configurable Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 632–642. IEEE, 2017. (cited on Page 10, 50, and 72)

Ya-Yunn Su, Mona Attariyan, and Jason Flinn. AutoBash: Improving Configuration Management with Operating System Causality Analysis. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–250. ACM, 2007. (cited on Page 28 and 73)

William N. Sumner and Xiangyu Zhang. Algorithms for Automatically Computing the Causal Paths of Failures. In *Proceedings of the International Symposium Foundations of Software Engineering (FSE)*, pages 355–369. Springer, 2009. (cited on Page 49, 50, 54, and 72)

William N Sumner and Xiangyu Zhang. Memory Indexing: Canonicalizing Addresses Across Executions. In *Proceedings of the International Symposium Foundations of Software Engineering (FSE)*, pages 217–226. ACM, 2010. (cited on Page 2, 47, 50, 55, and 56)

William N Sumner and Xiangyu Zhang. Comparative causality: Explaining the differences between executions. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 272–281. IEEE Press, 2013. (cited on Page 2, 4, 47, 49, 50, 54, 58, 60, 61, 62, 72, 73, and 76)

William N Sumner, Tao Bao, Xiangyu Zhang, and Sunil Prabhakar. Coalescing Executions for Fast Uncertainty Analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 581–590. ACM, 2011. (cited on Page 28)

Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review*, 45(3):10–14, 2012. (cited on Page 10)

Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys (CSUR)*, 47(1):6:1–6:45, 2014. (cited on Page 1, 8, 9, 46, and 75)

Thomas Thüm, Jens Meinicke, Fabian Benduhn, Martin Hentschel, Alexander von Rhein, and Gunter Saake. Potential Synergies of Theorem Proving and Model Checking for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 177–186, 2014. ACM. (cited on Page 10)

Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. Efficient Online Validation with Delta Execution. *ACM SIGARCH Computer Architecture News (SIGARCH)*, 37 (1):193–204, 2009. (cited on Page 73)

Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. A Classification of Product Sampling for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, 2018. (cited on Page 9)

Alexander von Rhein, Sven Apel, and Franco Raimondi. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. In *Proceedings of the Java Pathfinder Workshop*, 2011. (cited on Page 11, 21, 28, 32, 34, 36, 50, and 72)

Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Proceedings of the International Symposium on New Ideas, New Paradigms,*

and Reflections on Programming & Software (Onward!), pages 213–226. ACM, 2014. (cited on Page 16, 25, 26, and 77)

Dasarath Weeratunge, Xiangyu Zhang, William N. Sumner, and Suresh Jagannathan. Analyzing Concurrency Bugs Using Dual Slicing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 253–264. ACM, 2010. (cited on Page 49 and 50)

Mark Weiser. Program Slicing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 439–449, 1981. IEEE. (cited on Page 73)

Michael Weiss, Babak Esfandiari, and Yun Luo. Towards a Classification of Web Service Feature Interactions. *Computer Networks*, 51(2):359–381, 2007. (cited on Page 1)

Chu-Pan Wong, Jens Meinicke, Kästner, and Christian. Beyond Testing Configurable Systems: Applying Variational Execution to Automatic Program Repair and Higher Order Mutation Testing. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering - New Ideas and Emerging Results (ESEC/FSE-NIER)*. ACM, 2018a. (cited on Page 77)

Chu-Pan Wong, Jens Meinicke, Lukas Lazarek, and Christian Kästner. Faster Variational Execution with Transparent Bytecode Transformation. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 2018b. (cited on Page 2, 11, 15, 19, 20, 23, 24, 26, 27, 28, 29, 45, 48, 50, 58, 72, 73, 75, and 77)

W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering (TSE)*, 42(8):707–740, 2016. (cited on Page 49)

Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient Program Execution Indexing. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 238–248. ACM, 2008. (cited on Page 47, 50, 55, and 72)

Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, Dynamic Information Flow for Database-backed Applications. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2016. (cited on Page 2, 21, 27, and 71)

Pamela Zave. Software Requirements and Design: The Work of Michael Jackson, chapter Modularity in Distributed Feature Composition, pages 267–290. Good Friends Publishing Company, 2009. (cited on Page 1 and 7)

Andreas Zeller. Yesterday, my Program Worked. Today, it Does Not. Why? In *Proceedings of the International Symposium Foundations of Software Engineering (FSE)*, pages 253–267. ACM, 1999. (cited on Page 47)

Andreas Zeller. Isolating Cause-Effect Chains From Computer Programs. In *Proceedings of the International Symposium Foundations of Software Engineering (FSE)*, pages 1–10. ACM, 2002. (cited on Page 2, 4, 47, 49, 50, 54, 58, 60, 61, 62, 72, and 76)

Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering (TSE)*, 28:183–200, 2002. (cited on Page 47, 50, and 72)

Sai Zhang and Michael D. Ernst. Automated Diagnosis of Software Configuration Errors. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 312–321. IEEE, 2013. (cited on Page 51)

Sai Zhang and Michael D Ernst. Which Configuration Option Should I Change? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 152–163. ACM, 2014. (cited on Page 51)

Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta. Towards Locating Execution Omission Errors. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 415–424. ACM, 2007. (cited on Page 73)

Hamza Zulfiqa. Towards Detecting Unintended Feature Interactions. Master's thesis, TU Darmstadt, Germany, 2016. (cited on Page 71)

Ehrenerklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; verwendete fremde und eigene Quellen sind als solche kenntlich gemacht. Insbesondere habe ich nicht die Hilfe einer kommerziellen Promotionsberaterin/eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Ich habe insbesondere nicht wissentlich:

- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht Unterlassungs- und Schadensersatzansprüche der Urheberin/des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen kann. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht.

Magdeburg, den 7.1.2019

Jens Meinicke