University of Magdeburg

School of Computer Science

Master's Thesis

VarexJ: A Variability-Aware Interpreter for Java Applications

Author:

Jens Meinicke

December 3, 2014

Advisors:

Prof. Gunter Saake
Dipl.-Inform. Thomas Thüm
University of Magdeburg · Databases and Software Engineering

Prof. Christian Kästner
Carnegie Mellon University · Institute for Software Research

Meinicke, Jens:
VarexJ: A Variability-Aware Interpreter for Java Applications
Master's Thesis, University of Magdeburg, 2014.

Abstract

Many modern software systems can be customized to fulfill specific customer needs. Customization improves quality, extensibility, and usability. However, customization also comes with challenges for software analyses, because of the configuration space explosion. To analyze all configurations in isolation is expensive and often impractical. A main goal of current research on configurable systems is to provide new techniques to analyze all configurations. For some static analyses, such as type safety, this is already feasible by considering variability internally. When testing configurable software usually sampling strategies are used to test a subset of all configurations. These approaches might miss faults that are only contained in specific configurations and require redundant calculations. In this work, we use an approach where several configurations can be executed at once, while redundant calculations can be avoided, and thus effort for testing can be reduced. Based on previous work on variability-aware execution, we discuss how a Java Virtual Machine can be lifted to handle variability internally to execute all combinations of configuration options simultaneously. Specifically, we lifted the interpreter of Java Pathfinder. We show the variability-aware interpreter reduces time for testing of all configurations by orders of magnitude compared to testing of all program variants. We applied the interpreter to 10 configurable programs and gain a speed-up of up to 2,843 compared to brute-force execution. Variability-aware execution can lead to a new way of testing and analyzing configurable systems.

Acknowledgements

I would like to thank my advisor Prof. Christian Kästner who gave me the opportunity to visit the Carnegie Mellon University during my thesis. I thank anyone at the Carnegie Mellon University and in Pittsburgh who supported me and gave me a pleasant stay.

I also thank my advisors Thomas Thüm and Prof. Gunter Saake. Without their support and advice this work would have not been possible in this scope. In particular, I thank Thomas for his support during the last years.

Finally, I thank my family and friends for their moral support.

Contents

List of Figures **x**

List of Tables **xi**

List of Code Listings **xv**

1 Introduction **1**

2 Background: Programming with Conditional Values **5**
- 2.1 The Choice Calculus 5
- 2.2 Variational Data 6
- 2.3 Variational Programming 8

3 Variability-Aware Execution of Java Applications **13**
- 3.1 Problem Statement 13
- 3.2 Family-Based Analyses 16
- 3.3 Variability-Aware Execution 17
- 3.4 Java Virtual Machine 19
  - 3.4.1 Architecture 19
  - 3.4.2 Execution of Java Bytecode 21
- 3.5 Variability-Aware Execution of Java Bytecode 24
- 3.6 Summary 27

4 Implementation: Variability-Aware Execution with VarexJ **29**
- 4.1 Java Pathfinder 29
  - 4.1.1 Requirements and Discussion 30
  - 4.1.2 Architecture of Java Pathfinder 31
- 4.2 Fundamental Extensions for Variability-Aware Execution 33
  - 4.2.1 Incremental Lifting of Java Applications 34
  - 4.2.2 Method Frame 38
  - 4.2.3 Bytecode Instructions 40
  - 4.2.4 Scheduling 43
- 4.3 Specialized Extensions for Variability-Aware Execution 47
  - 4.3.1 Specification of the Configuration Space 47
  - 4.3.2 Peer Methods 49

4.3.3 Aggregated Results 50
4.4 Improvements of Variability-Aware Execution 53
4.4.1 Current Limitations 53
4.4.2 Optimizations 54
4.5 Summary 58

5 Evaluation **59**
5.1 Target Systems 59
5.2 Evaluation Framework 62
5.3 Efficiency of Variability-Aware Execution 66
5.4 Sharing and Interactions 71
5.5 Discussion 75
5.6 Summary 78

6 Related Work **79**

7 Conclusion ana Future Work **83**

Bibliography **87**

List of Figures

1.1 Brute-force analysis compared to variability-aware analysis. 3

2.1 Abstract representation of a tag tree with two features. 7

2.2 Abstract representation of a formula tree with three features. 8

3.1 Illustration of challenges for testing in general. 15

3.2 Illustration of challenges for testing of configurable systems. 16

3.3 Illustration of variability-aware testing solving the challenges of testing configurable software. 18

3.4 Platform and language-independent design of Java Virtual Machines. . 20

3.5 Overall architecture and elements of the Java Virtual Machine. 21

4.1 Interaction of layers in Java Pathfinder. 32

4.2 Top-level structure of Java Pathfinder, containing basic elements which need to be lifted. 33

4.3 Traces for product-based and variability-aware execution. 45

4.4 Illustration of the support of native method calls in Java Pathfinder. . . 49

5.1 Execution times for product-based and variability-aware testing. 67

5.2 Overhead of variability-aware execution compared to the average time for execution of one product with Java Pathfinder. 68

5.3 Instructions for variability-aware execution with VarexJ relative to instructions for product-based testing with Java Pathfinder. 69

5.4 Overhead of executed instructions for variability-aware execution compared to the execution of one product. 70

5.5 Measurements of maximal required memory for product-based and variability-aware testing. 71

5.6 Distributions of context sizes of executed instructions for each system. . . 72
5.7 Average distribution of the context size of executed instructions over all systems. 73
5.8 Maximal number of features required to define conditional values. The number of features of the system is shown in the braces after the name. 74
5.9 Maximal size of conditional values. 75

List of Tables

3.1 Traces and method frame states for the executions of bytecode instructions of the configurable example in Listing 3.3 and Listing 3.4. 26

5.1 Overview on configurable systems we use for evaluation of VarexJ. . . . 61

List of Code Listings

2.1 Example for alternative implementations of cubic functions. 5

2.2 Representation of alternative cubic functions using the choice calculus. 6

2.3 Representation of four alternative cubic functions with two dimensions. 6

2.4 Implementation of a tag tree for to represent conditional values. 8

2.5 Implementation of the map function for formula trees to modify conditional values. 9

2.6 Example for evaluation of conditional values with the map function. . . 10

2.7 Implementation of flat map for formula trees for context specific modifications of conditional values. 11

2.8 Example for conditional evaluation of conditional values with flat map. 11

2.9 Example to calculate the sum of two conditional values with a cross-product using fmap and map. 12

3.1 Example for challenges of testing in general. 14

3.2 Example for challenges of testing of configurable systems with defects. . 16

3.3 Configurable source code example with two program variants, depending on the value of the field `feature`. 23

3.4 Compiled Java Bytecode for the configurable example in Listing 3.3. . . 23

4.1 Example to illustrate the effort for lifting of a simple method. 34

4.2 Refactored example of Listing 4.1 supporting conditional values. 35

4.3 Transformation from conditional to concrete values for incremental lifting using runtime exceptions. 36

4.4 Example for incremental lifting of the example in Listing 4.1 to temporary avoid a complete refactoring. 36

4.5 Alternative refactorings to introduce a new parameter for the context. . 37

4.6 Simplified method frame implementation of Java Pathfinder. 39

4.7 Inefficient implementation for a variability-aware method frame. 39

4.8 Java Pathfinder implementation of the operand stack management instruction `swap`. 40

4.9 Efficient variability-aware stack frame implementation used by VarexJ. 41

4.10 Signature of Java Bytecode instructions in Java Pathfinder. 41

4.11 Signature of variability-aware Java Bytecode instructions in VarexJ. . . 41

4.12 Example bytecode instruction IADD in Java Pathfinder. 42

4.13 Variability-aware implementation of IADD in VarexJ. 42

4.14 Simplified default scheduling mechanism of Java Pathfinder. 43

4.15 Simplified variability-aware scheduling mechanism of VarexJ. 44

4.16 Example configurable method for variability-aware scheduling. 44

4.17 Source code example with multiple return statements. 46

4.18 Compiled example with multiple return instructions. 46

4.19 Example for method invocation on different objects. 47

4.20 Annotated field to introduce conditional values into the interpreter. . . 48

4.21 Example for specifying constraints with a method that checks the validity of the current feature selection, to execute only valid configurations. . . 48

4.22 Original implementation of the peer method for `java.lang.Math.max` in Java Pathfinder. 50

4.23 Lifted peer method for `java.lang.Math.max` in VarexJ with conditional parameters, conditional return type, and a context. 50

4.24 Aggregation of errors with VarexJ. 51

4.25 Aggregation of console outputs with VarexJ. 52

4.26 Example for possible optimization of variability-aware executions with sharing among objects. 54

4.27 Example for possible optimization of variability-aware executions with sharing among branches. 55

4.28 Example for possible optimization of variability-aware execution with sharing among methods. 56

5.1 Example file representing configurations for a target system. 62

5.2 Example file for selection of a configuration and execution of the application. 63

5.3 Example file for measuring of execution times. The time is printed to the console which can be read by our framework. 64

5.4 Example statistics calculated by Java Pathfinder. The statistics we reuse are highlighted with gray. 65

1. Introduction

Many modern software systems are designed to be highly customizable to increase flexibility, quality, and security, and to match specific customer needs [Clements and Northrop, 2001; Pohl et al., 2005; Apel et al., 2013a]. There are several techniques to introduce customization into a program, such as configuration files, command-line options, preprocessors, plug-ins, aspect-oriented programming [Kiczales et al., 1997], and feature-oriented programming [Prehofer, 1997]. Each configuration option can define a specific property of the program. A configuration option can be any kind of value, such as numerical and boolean values. In this work, we refer to binary options (a.k.a. features) only.

Configurable systems are designed to match specific customer needs, but customization also comes with challenges for software analysis. A system with n optional features contains up to 2^n different program variants. Thus, even for a small amount of features, analyzing all program variants in isolation can get impractical. For a program with 320 optional and independent features there are already as many variants as atoms in the universe. However, configurable systems can contain thousands of options.

To analyze all program variants in isolation is inefficient. To analyze configurable systems anyway, there are several approaches that select specific program variants, known as *sampling* strategies [Nie and Leung, 2011; Apel et al., 2013a; Thüm et al., 2014a]. Sampling strategies attempt to detect as many faults as possible by analyzing a small subset of configurations only. A common strategy is to analyze the set of program variants that are actually used in practice. Another common strategy is to analyze a product that contains all features (a.k.a. allyes or full configuration [Dietrich et al., 2012; Liebig et al., 2013]). Because some features might be exclusive (e.g., features for different operation systems), the more general approach of *feature coverage* selects products such that all features are selected at least once [Apel et al., 2013a]. Because the selection of features might also deactivate some code fragments, the sampling strategy *feature-code coverage* (a.k.a. configuration coverage) selects products, such that any code fragment is contained in least one product [Sincero et al., 2010; Tartler et al., 2012]. These sampling strategies are efficient for detecting of defects that are caused by the implementation of a specific feature, or a specific code fragment. They drastically reduce the set of products that need to be analyzed.

In configurable systems many failures are caused by *feature interactions* [Calder et al., 2003; Nhlabatsi et al., 2008]. *Combinatorial interaction testing* is a sampling strategy which aims to detect failures caused by interactions of program parameters [Cohen et al., 1997, 2007]. To cover and test all combinations of τ features is called τ-way interaction testing (i.e., to cover all combinations of 2 parameters, $\tau = 2$). To only cover τ-way interactions reduces the number of configurations to test, and even scales for a high amount of features. Furthermore, 6-way interaction testing is argued to cover most defects [Kuhn et al., 2004]. Combinatorial interaction testing uses covering arrays that contain all required parameter combinations. Because the generation of covering arrays gets more expensive for higher τ, combinatorial interaction testing usually only scales for up to $\tau = 6$, what is already high, expensive, and uncommon in practice [Petke et al., 2013]. However, some defects might be caused by interactions of more than 6 features [Liebig et al., 2010; Apel et al., 2013c]. Thus, combinatorial interaction testing might miss faults caused by interactions of higher feature interactions. Especially, to ensure properties, such as safety and security, it is necessary to detect also defects caused by interactions of more than 6 features.

For some applications it is not necessary to detect all defects. For example a configurable system with a fixed set of products only needs do detect defects in this set, such as HP's printer firmware with a limited amount of printers. Anyhow, even if the set of systems which needs to be analyzed is limited, tests need to be applied to all these products. As all these variants share parts of the program, the tests execute parts of the program redundantly. Thus, the effort for testing even a small subset of all program variants is unnecessarily high.

Sampling strategies analyze products in isolation. When a failure occurs on a specific program variant, it is a difficult task to aggregate the result to a specific subset of products. Assume that, the selection of feature A and feature B result in a failure. To aggregate such statements out of the result of sampling is difficult, even for interaction testing [Thüm et al., 2014a]. We summarize pitfalls of sampling strategies as follows:

- Analysis of subset of products, and thus detection of subset of defects only.
- Redundant calculation on several variants.
- No aggregated results for defects.

The goal of current research for configurable systems and software product lines is to provide new analyses strategies that can analyze a configurable software system with the same results as for brute-force testing of all possible program

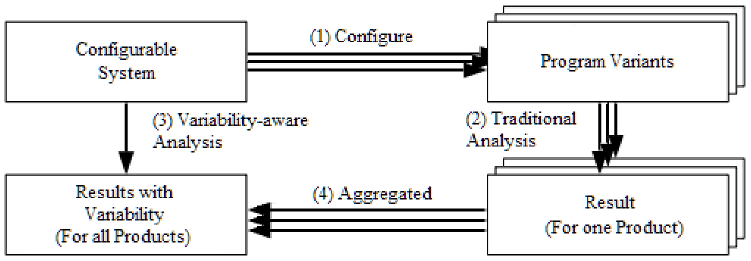

Figure 1.1: Brute-force analysis compared to variability-aware analysis.

variants [von Rhein et al., 2013]. Such analyses are known as variability-aware or family-based analyses [Liebig et al., 2013; Thüm et al., 2014a]. Variability-aware analysis takes variability into account during analysis and can detect all failures caused by feature interactions independent of the degree of the interaction (i.e., interactions among all features can be detected). Variability-aware analysis is efficient because redundant calculations can be avoided, thus even systems such as Linux with more than 10,000 configuration options can be analyzed [Tartler et al., 2011; Liebig et al., 2013].

In Figure 1.1, we illustrate brute-force analysis compared to variability-aware analysis. When analyzing all program variants, they need to be configured and eventually generated first ((1) in Figure 1.1). In the second step, all these program variants need to be analyzed, one after another (2). In contrast, variability-aware analysis can operate directly on the configurable system or an abstraction thereof [Thüm et al., 2014a] (3). Variability-aware analyses provide the aggregated results as if all variants (4). Furthermore, the result of variability-aware analysis should specify the subset of program variants for each defect, to identify the corresponding feature interactions.

Previous work for variability-aware execution showed that sharing of program executions can reduce the effort for testing [Kim et al., 2012; Kästner et al., 2012b; Apel et al., 2013d]. With shared execution parts of the code can be executed among several configurations, and thus effort for testing can be reduced [Kim et al., 2012]. With the JavaPathfinder (JPF) extension JPF-BDD, executions can be shared efficiently while the effort compared to the core implementation can be reduced by orders of magnitude, because many execution paths can be joined [Kästner et al., 2012b; Apel et al., 2013d]. Both approaches for variability-aware execution try to reduce the analysis effort by sharing executions among program variants. However, these approaches are only able to handle different values for configuration options. When other values differ (e.g., a value of a field or local variable differs for different feature selections), these approaches need to split execution paths and execute the same code several times.

A variability-aware interpreter is able to execute instructions on different values at the same time [Kästner et al., 2012b; Nguyen et al., 2014]. Kästner et al. [2012b] developed a prototypical interpreter for a WHILE language. Based on this, Nguyen et al. [2014] developed an interpreter for the plug-in application WorldPress. A variability-aware interpreter maximizes sharing among program variants and thereby reduces the effort for testing by orders of magnitude compared to brute-force testing [Kästner et al., 2012b; Nguyen et al., 2014]. Furthermore, variability-aware execution eases the detection of feature interactions, because program variables can be directly mapped to specific configuration spaces.

Previous work on variability-aware interpretation showed promising results. However, both implementations come with major restrictions. The interpreter for the WHILE language is a proof of concept and only works for a toy language [Kästner et al., 2012b]. The PHP interpreter is more powerful, but because of the complex language design of PHP, the variability-aware interpreter is incomplete and can only execute the WorldPress application [Nguyen et al., 2014].

Contribution

Current implementations of variability-aware interpreters have major limitations. The goal of this thesis is to present a more general implementation of a variability-aware interpreter. In particular, we want to support a complete programming language. Java Bytecode is a language with a clear specification and a manageable amount of instructions. We show how a Java Virtual Machine (JVM) can be transformed to handle variability internally. Specifically, we developed our variability-aware interpreter VarexJ based on the interpreter of JPF [Havelund and Pressburger, 2000; Visser et al., 2003].

As variability-aware analyses should provide aggregated results [Thüm et al., 2014a], we implemented and present our solutions for aggregated console outputs and exceptions. We discuss several optimizations that can be applied to variability-aware interpreters. Furthermore, we gained new insights on sharing and redundancies in executions. In particular, we proposed several new types of sharing that can improve variability-ware analyses and can lead to a new area of research.

We applied VarexJ to 10 configurable systems, to show the scalability of the approach for higher degrees of interactions than occurring in plug-in systems and to show that the interpreter can execute arbitrary programs. With these systems, we show that the time for testing can be reduced by orders of magnitude. We compare our interpreter to product-based testing with the core implementation of JPF with a speed-up of up to 2843. We also compare our implementation with the JVM from Oracle, because JPF has a lot of overhead compared to a common JVM. Finally, we show how a variability-aware interpreter can efficiently analyze interactions in configurable software.

Structure of the Thesis

This thesis is structured as follows. We introduce the reader into basic concepts of configuration options and programming with conditional values in Chapter 2. In Chapter 3, we discuss the state-of-the-art of variability-aware testing and show how to variability-aware execution can be applied to Java Bytecode. We present our implementation of the variability-aware interpreter in Chapter 4. In Chapter 5, we evaluate our interpreter and discuss the results. We present related work in Chapter 6. We conclude and discuss future work in Chapter 7.

2. Background: Programming with Conditional Values

Variability-aware execution requires an efficient representation of variability. In this chapter, we discuss basic techniques for such a representation. In Section 2.1, we introduce the choice calculus, an abstract language for the representation of variability. In Section 2.2, we explain variational programming, a concrete language to calculate with variability based on the concepts of choice calculus.

2.1 The Choice Calculus

Many modern software systems use some sort of variation. To efficiently deal with variability in software and for a distinct representation of variability, Erwig and Walkingshaw [2011] introduced the *choice calculus*. The choice calculus is a language to represent software variation. The choice calculus is a representation of variation that is independent of the implementation of the variation and programming language, and thus eases the research transfer between research fields [Erwig and Walkingshaw, 2011]. The choice calculus represents a mapping of a tags (i.e., alternative features such as F or G) to their corresponding implementations (e.g., if F then f(x) else if G then g(x)).

```
cube x = x * x * x
cube x = x ^ 3
```

Listing 2.1: Example for alternative implementations of cubic functions.

To explain the basic principles of the choice calculus, we use a simple cubic function. The example in Listing 2.1 shows two alternative implementations of `cube` for calculation of the cube of a given value `x`. The implementations have two alternative solutions. However, both implementations share commonalities, which can be used for an efficient representation with the choice calculus. In Listing 2.2, we illustrate a representation of the cubic functions using the choice calculus. With the keyword **dim**, the dimensions and their elements are specified. In the example, **dim** `Impl<Times, Exp>` specifies that the dimension `Impl` has two alternative tags: `Times` and `Exp`. A *tag* defines an alternative solution for one dimension. The keyword **in**, defines the corresponding scope of the expression. In the cubic example, **in** refers to the function `cube`. Each tag of a dimension can be mapped to a corresponding implementation. Furthermore, all tags of a dimension are alternative, thus only one tag can be active at the same time.

```
dim Impl<Times, Exp> in
cube x = Impl<x * x * x, x ^ 3>
```

Listing 2.2: Representation of alternative cubic functions using the choice calculus.

If a second dimension is introduced to the example of Listing 2.1, the whole code needs to be cloned multiple times, such that all combinations of all dimensions have a corresponding representation. For example, assume that a second dimension specifies whether the function returns absolute values. With the second dimension, there are four alternative implementations. With the choice calculus, this can be expressed as shown in Listing 2.3.

```
dim Abs<Absolute, Relative> in
dim Impl<Times, Exp> in
cube x = Abs<abs(Impl<x * x * x, x ^ 3>), Impl<x * x * x, x ^ 3>>
```

Listing 2.3: Representation of four alternative cubic functions with two dimensions.

2.2 Variational Data

In this section, we explain how the ideas of the choice calculus can be realized to directly calculate with variability in software. First, we introduce into the concept of conditional values. Then, we discuss the two data structures tag tree and formula tree as representation of conditional values.

Conditional Values

A configurable system provides several *features*. A *configuration* represents a specific selection of these features. The set of all possible combinations of features is called *configuration space*. Because some combinations of features might be invalid, there can be *constraints* among features (e.g., if feature A is selected, than feature B has to be selected, too). Such constraints can be defined with a *feature model* [Kang et al., 1990]. A configuration is *valid* if it fulfills all constrains. A *context* is a specific set of configurations (e.g., all configurations where feature A is selected). A *conditional value* is a mapping of concrete values to the corresponding contexts. There are several data structures which implement conditional values, such as tag trees, formula trees, and formula maps [Walkingshaw et al., 2014]. In this section, we describe how tag trees and formula trees can be used to represent conditional values. Furthermore, we discuss how to compute with these data structures (e.g., to multiply two conditional values).

Tag Tree

A tag tree represents conditional values as a mapping from tags to values, where each tag represents a separate node of the tree [Erwig and Walkingshaw, 2011]. In Figure 2.1, we illustrate the representation of a tag tree with two features. Each node represents only one feature. A branch represents the selection of the feature, which can be either yes or no (i.e., selected or deselected). The leafs of the tree represent the concrete values.

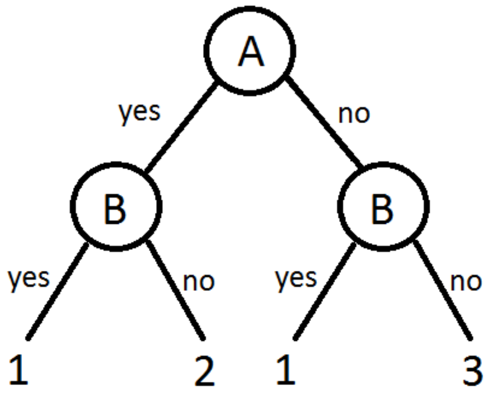

Figure 2.1: Abstract representation of a tag tree with two features.

To understand how tag trees can be used to represent conditional values, we show an implementation in Listing 2.4. The shown implementations for tag trees and formula trees are based on a library of TypeChef [Liebig et al., 2013; Kästner et al., 2012a, 2011]. The interface `Conditional` represents conditional values of type `T`. The class `Choice` represents the nodes of the tree. A `Choice` contains a tag and conditional values which depend on the tag. The class `One` represents concrete values (i.e., the leafs of the tree). The context of a concrete value is the conjunction of the corresponding choice tags. In the example of Figure 2.1, the context for the value 2 is $A \wedge \neg B$. Tag trees can be used to represent conditional values. However, a tag tree is inefficient if a conditional value depends on several feature selections, because each feature selection needs a separate choice, the depth of the tree grows with every feature. Thus, the tag tree is likely to contain several redundant values, what is memory inefficient, and requires redundant calculations [Walkingshaw et al., 2014].

Formula Tree

In contrast to a tag tree, a formula tree allows propositional formulas instead of atomic tags [Walkingshaw et al., 2014]. Instead of expressing a context with nested choices, formula trees allow to define them directly with formulas in choices (e.g., A∨B). Formula trees are a more flexible representation of conditional values than tag trees. However, to reason about valid configurations requires to solve a satisfiability problem (e.g., with SAT solvers or BDDs) [Walkingshaw et al., 2014]. In Figure 2.2, we illustrate an abstract representation of a formula tree.

```
interface Conditional<T> {}
class One<T> implements Conditional<T> {
    T value;
    One(T value) { ... }
}
class Choice<T> implements Conditional<T> {
    Tag t;
    Conditional<T> yes, no;
    Choice(Tag t, Conditional<T> yes, Conditional<T> no) { ... }
}
```

Listing 2.4: Implementation of a tag tree for to represent conditional values.

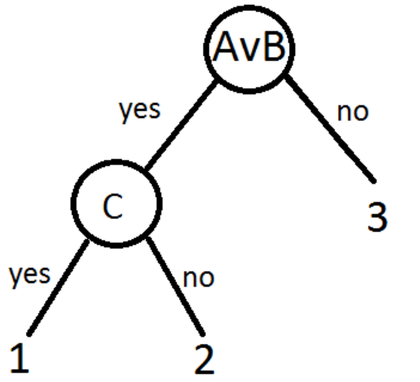

Figure 2.2: Abstract representation of a formula tree with three features.

2.3 Variational Programming

With tag trees and choice trees, we described two data structures that encode conditional data. To calculate with such conditional values, functions that can evaluate conditional values are necessary [Erwig and Walkingshaw, 2013]. To illustrate such an evaluation, we use following notion:

$$One(y) \bullet f(x) \Rightarrow One(f(y))$$

$$Choice(A, a, b) \bullet f(x) \Rightarrow Choice(A, a \bullet f(x), b \bullet f(x))$$

A function f can be directly applied to the value encapsulated in a *One*, or recursively to the elements of a *Choice*. $\bullet$ indicates that the function on the right side is applied to the conditional value on the left side. The arrow represents the evaluation of the function. For example, if the values of a conditional value should be incremented by 2 (for brevity, we only show y instead of $One(y)$):

$$Choice(A, 1, 2) \bullet (x + 2) \Rightarrow Choice(A, 3, 4)$$

The map function (known from functional programming) modifies conditional values by applying a given function to *all* elements of the choice [Erwig and Walkingshaw, 2013]. In Listing 2.5, we show an example implementation of a map function for conditional values. The input of the method map is a function f, which evaluates the value of type T to a value of type U. In the examples, we use Java 8 syntax, to take advantage of anonymous functions. The interface Function is introduced in Java 8 as assignment target for a lambda expression or a method reference. With the method apply the function f can be applied to a given parameter. The map implementation of the class Choice passes the function f to both conditional values. In the map implementation of One, the function f is applied to the concrete value.

```
interface Conditional<T> {
    Conditional<U> map(Function<T, U> f);
}
class One<T> implements Conditional<T> {
    T value;
    <U> Conditional<U> map(Function<T, U> f) {
        return new One<>( f.apply(value) );
    }
}
class Choice<T> implements Conditional<T> {
    FeatureExpr ctx;
    Conditional<T> yes, no;
    <U> Conditional<U> map(Function<T, U> f) {
        return new Choice<>(ctx, yes.map(f), no.map(f) );
    }
}
```

Listing 2.5: Implementation of the map function for formula trees to modify conditional values.

The function f evaluates a value of type T to an arbitrary object of type U. Thus, the elements of the evaluated choice can have another type than the elements of the original choice. For example, a transformation from Integer to Boolean is also possible:

$$Choice(A, 1, 2) \bullet (x\%2 == 0) \Rightarrow Choice(A, false, true)$$

In Listing 2.6, we illustrate how map can be used to modify conditional values. In the lines starting with the $-sign we show the current values of c. In the first part, the increment function is applied to all elements of the conditional value c (Lines 1 to 4). In the second part the type of conditional values is changed after the modulo-function is applied (Lines 7 to 9). Furthermore, the example shows that the structure of the formula tree does not change after map is applied.

The map function is a simple way to calculate with conditional values. However, to apply a function only for a specific context, the function f needs to evaluate to Conditional<U>. If such a function is applied with map, then the evaluated

```
Conditional c = new Choice(A, 1, 0);
c = c.map((x) -> {
    return x + 2;
});
$ c: Choice(A, 3, 2)

c = c.map((x) -> {
    return x % 2 == 0;
});
$ c: Choice(A, false, true)
```

Listing 2.6: Example for evaluation of conditional values with the map function.

conditional value is of type `Conditional<Conditional<U»`. The reduction to `Conditional<Conditional<U»` is called *flattening*. The corresponding function which applies a functions `f` only for a specific context is called *flat map*. To indicate that a function is only applied for a specific context, we add the context to the circle as $\overset{ctx}{\bullet}$. We illustrate context-specific evaluations of conditional values as follows:

$$One(y) \overset{A}{\bullet} f(x) \Rightarrow Choice(A, f(y), y)$$

$$Choice(B, a, b) \overset{A}{\bullet} f(x) \Rightarrow Choice(B, a \overset{A}{\bullet} f(x), b \overset{A}{\bullet} f(x))$$

When a function $f(x)$ is applied to a One for a context A, then a new Choice is created that represents the evaluated value for A and the old value for $\neg A$. When the function $f(x)$ is applied to a Choice then the function is applied to its values recursively. Because the result might contain invalid feature combinations (e.g., $A \wedge \neg A$), such values can be removed afterward by solving the insatiability of the corresponding expressions. We illustrate calculations with flat map with concrete values in the following examples:

$$One(0) \overset{A}{\bullet} (x+1) \Rightarrow Choice(A, 1, 0)$$

$$Choice(A, 1, 0) \overset{A}{\bullet} (x+1) \Rightarrow Choice(A, 2, 0)$$

$$Choice(B, 1, 0) \overset{A}{\bullet} (x+1) \Rightarrow Choice(B, Choice(A, 2, 1), Choice(A, 1, 0))$$

The function *flat map* can apply a function which returns a conditional value. In Listing 2.7, we illustrate an implementation of flat map (`fmap`) for formula trees. The only difference of `fmap` to `map` is that the function `f` returns a conditional value directly, thus the implementation of `One` does not need to encapsulate the result in a new `One`.

To understand how the implementation of `fmap` can apply a context-specific function, we use the example in Listing 2.8. In the example, we apply a function that returns a choice representing the incremented value if A is selected, and the old value otherwise

```
interface Conditional<T> {
    <U> Conditional<U> fmap(Function<T, Conditional<U> > f); }
class One<T> implements Conditional<T> {
    T value;
    <U> Conditional<U> fmap(Function<T, Conditional<U> > f){
        return f.apply(value) ; }
}
class Choice<T> implements Conditional<T> {
    CTX ctx;
    Conditional<T> yes, no;
    <U> Conditional<U> fmap(Function<T, Conditional<U> > f){
        return new Choice<>(ctx, yes.fmap(f), no.fmap(f) ); }
}
```

Listing 2.7: Implementation of flat map for formula trees for context specific modifications of conditional values.

```
 1 Conditional c = new One(0);
 2 c = c.fmap((x) -> {
 3     return new Choice(A, new One(x + 1), new One(x));
 4 });
 5 $ c: Choice(A, 1, 0)
 6
 7 c = c.fmap((x) -> {
 8     return new Choice(A, new One(x + 1), new One(x));
 9 });
10 $ c: Choice(A, Choice(A, 2, 1), Choice(A, 1, 0))
11 $ c -> Choice(A, 2, 0)
```

Listing 2.8: Example for conditional evaluation of conditional values with flat map.

(Lines 3 and 8). First, we apply the function to a concrete value `c` (i.e., a `One`). The result is a choice with the incremented value for the context A and the old value otherwise. If we apply the same increment function a second time to `c`, the result is a nested choice, because the function is applied to all elements of `c`. The result contains elements for invalid expressions (e.g., for $A \wedge \neg A$). To remove such invalid expressions, SAT solvers or BDDs can be used by solving the satisfiability problem. Furthermore, constraints of feature models can be used to remove values for invalid configurations. After all invalid entries are removed; the result is a `Choice` where the original element is incremented twice if A is selected. To remove invalid entries and to reduce the number of duplicates always comes with additional effort [Walkingshaw et al., 2014].

When calculating with conditional values it is often necessary to apply multiple conditional values to each other. For example, when two conditional values are added, the result is the cross product of all valid combinations:

$$Choice(A, 0, 1) \bullet (x + Choice(B, 2, 3)) \Rightarrow Choice(A, Choice(B, 2, 3), Choice(B, 3, 4))$$

```
Conditional c1 = new Choice(A, new One(0), new One(1));
Conditional c2 = new Choice(B, new One(2), new One(3));
Conditional sum = c1.fmap((x) -> {
    return c2.map((y) -> {
        return x + y;
    });
});
$ sum: Choice(A, Choice(B, 2, 3), Choice(B, 3, 4))
```

Listing 2.9: Example to calculate the sum of two conditional values with a cross-product using fmap and map.

In Listing 2.9, we illustrate how the sum of two conditional values `c1` and `c2` can be calculated using a cross product. To access all values of `c1` we use the function `fmap`. To apply all values of `c1` to c2, we can use the function `map`. Finally, the values of `c1` are added to the values of `c2`. If the sum should only be calculated for a specific context, `fmap` needs to be used instead of `map`.

3. Variability-Aware Execution of Java Applications

In this chapter, we discuss the general concepts of variability-aware execution of Java applications as basis for our development of a variability-aware interpreter for Java Bytecode. In Section 3.1, we present the challenges for testing of configurable systems. In Section 3.2, we explain family-based analysis, a general strategy for efficient analysis of configurable systems. As solution for the challenges of testing configurable systems, we discuss variability-aware execution, and the approach of a variability-aware interpreter in Section 3.3. As basis for a variability-aware interpreter for Java Bytecode, we explain the general architecture and concepts of a JVM in Section 3.4. In Section 3.5, we combine execution of Java Bytecode with a JVM and the strategy of variability-aware execution, to discuss our approach of a variability-aware interpreter for Java Bytecode. This chapter can be used as general guideline to lift a Java Bytecode interpreter for variability-aware execution. We discuss the overall procedure how a variability-aware interpreter for Java works, but we hide any implementation details. We describe our implementation later in Chapter 4 that contains specific solutions for challenges of variability aware-execution.

3.1 Problem Statement

Testing configurable systems comes with several difficulties. To test all configurations individual is not efficient, because the number of program variants to test increases exponential to the number of features. Constraints between features can reduce the configuration space, but the number of variants is often still too large to test all individually. Several sampling strategies [Nie and Leung, 2011; Apel et al., 2013a] were proposed to reduce the number of variants to a subset with a high probability to detect faults, such as τ-wise sampling [Cohen et al., 1997]. However, to test only a subset of all products might miss defects, which could be detected if the program variant which contains the defect would be tested. Furthermore, testing several program variants requires redundant calculations, because the variants share code. Finally, testing individual systems can only give results that correspond to one specific configuration. To aggregate the results for all configurations to detect the specific interaction of features that cause the defect is hard.

Software testing in general has similar difficulties: It can only test a subset of possible inputs, the system behaves different for different inputs, and testing several inputs requires redundant calculations. In contrast to testing of systems with arbitrary and virtually endless inputs, testing configurable systems has the advantage that the number of features and their states (i.e., selected or deselected) is limited. Consequently, also the number of configurations is limited. However, the number of configurations still gets huge, because a system usually has many features. Thus, testing all configurations individually is still not efficient or possible. In this work, we use the properties configurable systems to have a fixed number of options (i.e., features) with only two states and a fixed number of combinations thereof (i.e., configurations) to provide efficient testing for configurable systems.

In Listing 3.1, we show an example method to illustrate the challenges of testing. The method `getDigits` should return an array that contains the first `n` digits, but the method also contains two defects. A goal of testing is to find input values that cause these defects. The first defect is caused by negative input values, because the array `digits` can only be initialized with a positive size. The second defect is caused by values larger than 10, because the method `charAt` gets out of bounds.

```java
static char[] getDigits(int n) {
    char[] digits = new char[n];
    for (int i = 0; i < n; i++) {
        digits[i] = "0123456789".charAt(i);
    }
    return digits;
}
```

Listing 3.1: Example for challenges of testing in general.

We visualize these challenges of testing the method `getDigits` in Figure 3.1. The figure shows how a system is tested for multiple inputs. The areas inside the valid inputs are unknown. They represent inputs which cause no defect {x >= 0 & x <= 10}, inputs for negative char array initialization {x < 0} (Defect A), and inputs for the out of bounds exception {x > 10} (Defect B). The goal of testing is to detect a value in each area which causes a defect. Good tests should reveal as many defects as possible with minimal effort [Ammann and Offutt, 2008]. Hence, the system is usually only tested for a small subset of values. Because the system is executed several times, many executions have to be done redundantly, such as the array initialization. Each input value results in a specific output. The values 5 and 10 cause no defects, but require redundant calculations. The input value -1 causes the defect A. However, that the defect is caused by smaller values is still unknown. Furthermore, the defect B stays undetected. To find defect B, the system has to be tested for more input values (e.g., for 11), which increases the effort and time to test. Testing can efficiently show the validity of a system for a specific test, but to find tests and input values to detect defects is challenging.

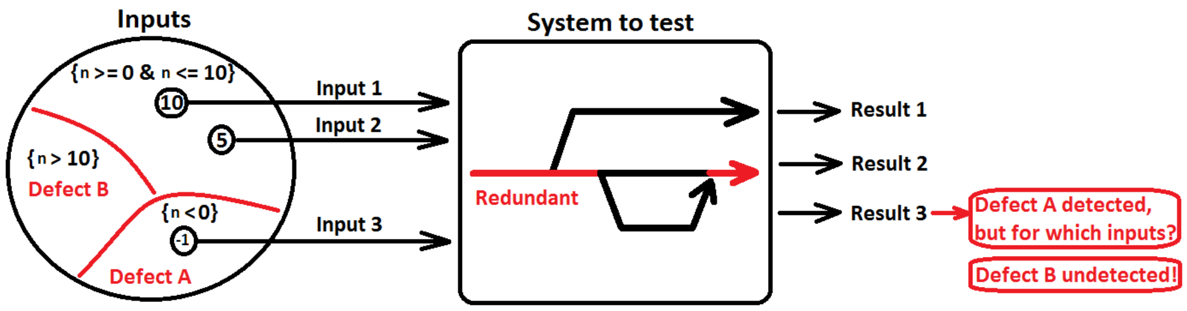

Figure 3.1: Illustration of challenges for testing in general.

Testing configurable systems shares the same challenges as testing in general. In addition to general testing, testing of configurable systems also needs to detect interactions and selections of features that cause a defect. Product-based analysis analyzes a configurable system for multiple configurations in isolation [Thüm et al., 2014a]. Product-based analysis is still common because it can reuse existing tools from single system engineering and does not require specialized tools. As not all program variants can be tested, sampling strategies [Apel et al., 2013a; Nie and Leung, 2011] try to find configurations which are likely to cause a defect.

To illustrate the challenges of testing configurable systems we reuse the example method `getDigits` in Listing 3.2. To focus on configuration options, we replace the parameter `n` by a fixed array size of 10. Furthermore, we added the two features `LETTERS` and `REVERSE`. If the feature `LETTERS` is true, the method returns letters instead of digits. The feature `REVERSE` reverses the array. Again the example contains two defects. The first defect is triggered when `LETTERS` is true. The method `charAt` throws an out of bounds exception, because the string containing the letters only contains 9 signs. The second defect is caused by the implementation of the reversing. Instead of copying the array `digits` in Line 10, the array `old` is only a reference. Thus, the implementation reads overwritten chars and returns {9,8,7,6,5,5,6,7,8,9} instead.

In Figure 3.2, we visualize product-based testing and its challenges with reference to the example of Listing 3.2. In the set of all feature selections only a subset represents valid configurations (e.g., `LETTERS` and `REVERSE` might not be allowed). This subset is defined with constraints among features, or represents practically used configurations. Still the tests for these configurations require redundant effort (e.g., the initialization of the array `digits`). The program variants for Configuration 1 and 2 do not cause defects, but require redundant effort. The test applied to Configuration 3 (e.g., for $LETTERS \wedge \neg REVERSE$) detects the defect A (the missing letter J in Line 5), but only by the outcome it is not clear for which features or interaction thereof. Furthermore, the defect B stays undetected (the wrong copy of digits).

```java
static boolean LETTERS = getFeature("LETTERS");
static boolean REVERSE = getFeature("REVERSE");
static char[] getDigits2() {
    char[] digits = new char[10];
    String values = LETTERS ? "ABCDEFGHI" : "0123456789";
    for (int i = 0; i < digits.length; i++) {
        digits[i] = values.charAt(i);
    }
    if (REVERSE) {
        char[] old = digits;
        for (int i = 0; i < digits.length; i++) {
            digits[i] = old[digits.length - i - 1];
        }
    }
    return digits;
}
```

Listing 3.2: Example for challenges of testing of configurable systems with defects.

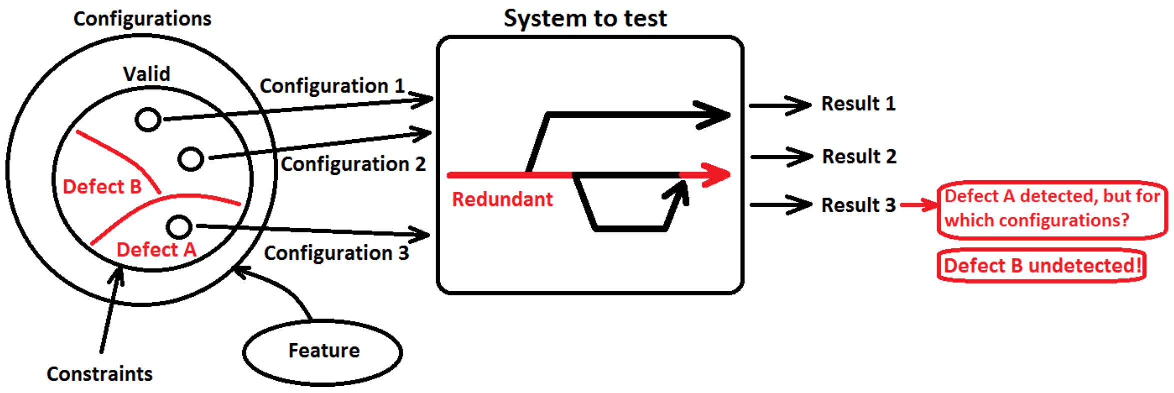

Figure 3.2: Illustration of challenges for testing of configurable systems.

3.2 Family-Based Analyses

To introduce into our solution for the challenges with product-based testing, we first explain a general strategy for efficient analyses of configurable systems. Besides product-based analyses there are several more strategies namely, feature-based analysis, family-based analysis and combinations of strategies (e.g., family-product-based) [Thüm et al., 2014a]. Feature-based analysis considers the implementation of each feature in isolation to detect defects in the implementation of single features. This strategy is efficient because it concentrates on small parts of the system and thereby reduces redundant effort. However, this analysis cannot predict defects caused by interactions of features, and consequently might miss defects. The family-based (a.k.a. variability-aware) strategy analyzes the whole system or a model thereof at once. Family-based analysis tools consider the variability of the program during analysis. Thus, family-based analysis can analyze all program variants at once, thereby reduce the analysis effort by sharing calculations and can return aggregated results.

Family-based analysis gives solutions for the described problems with product-based analysis. On the other hand, product-based analysis can reuse any analysis tool from single system engineering without a need for adjustments. Furthermore, analyzing one single configuration is much faster than a family-based analysis. In contrast to product-based analyses, family-based analyses often require specialized tools which can handle variability internally or require a specific encoding or model of the system [Thüm et al., 2014a]. It is challenging to provide sound analysis tools with equivalent analyses as analysis of all systems with tools from single system engineering.

There are several strategies to provide tools for family-based analyses. The simplest strategy is to reuse existing tool which are already able to handle variability [Thüm et al., 2014a], such as model checker [Kästner et al., 2012b] or theorem prover [Thüm et al., 2012]. To apply these tools often a specialized encoding of the system is required, namely metaproduct or product simulator [Thüm et al., 2014a]. Because these tools can only handle variability to some point, and are not optimized for sharing, there is potential left for more efficient analyses. Another strategy is to build a completely new tool which is specialized for variability. Such tools are efficient, but require high effort for development. Furthermore, they are often only able to detect a subset of defects compared to the tools of single-system engineering, especially because of the additional effort for handling of variability. In our work, we use another strategy. We lift an existing tool to handle variability to reuse its functionality. Specifically, we lift an existing JVM to handle conditional values for variability-aware executions to test configurable Java Bytecode programs. To lift existing analysis tools can also be useful for analyses beyond executions, especially because the analyses techniques do not need to be reimplemented.

3.3 Variability-Aware Execution

In this section, we discuss how configurable systems can be tested efficiently with variability-aware execution. Variability-aware execution can solve the discussed disadvantages of product-based testing; it executes all valid program variants, it reduces redundant effort, and thus reduces time for testing, and it provides aggregated results.

The general idea of variability-aware execution is to share executions among configurations. With sharing, the time for execution is reduced compared to product-based testing. Variability-aware execution can be so efficient that testing of all configurations which would take years can be done in minutes [Nguyen et al., 2014]. To improve variability-aware testing, sharing needs to be maximized, to reduce redundant calculations. Sharing of executions is already supported by tools, such as shared execution [Kim et al., 2012] and JPF-BDD [Kästner et al., 2012b]. However, both implementations leave potential for sharing and many executions are still executed redundantly.

In Figure 3.3, we illustrate variability-aware testing with reference to the previous example in Listing 3.2. Again the set of all feature combinations can be reduced by constraints. Instead of concrete configurations, the system gets the features and their constraints as input (i.e., the tool assumes both selections of the features, such as LETTERS and REVERSE). These constraints are used to reason about valid feature combinations, and thus about valid executions (i.e., the tool reasons about configurations spaces instead of concrete configurations). During variability-aware execution, parts of the program can be shared among several configurations to reduce the effort for testing (e.g., the array digits is only initialized once). The outcome of variability-aware analysis is a mapping of a specific result to a subset of valid configurations {SET 1, SET 2, SET 3}. For example, all result for the configurations where LETTERS is selected (SET 2) is an out of bounds exception caused by the method charAt (Defect A). Furthermore, it can detect defects for specific feature combinations, which might be missed with product-based strategies.

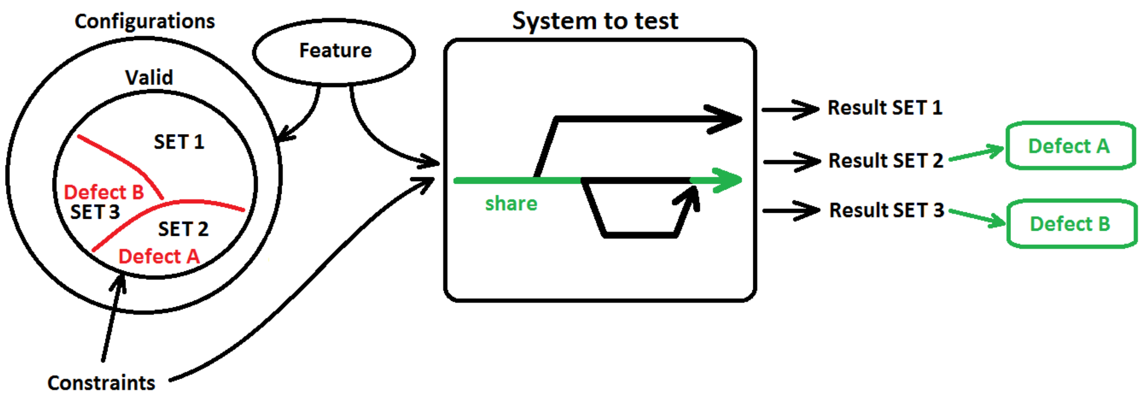

Figure 3.3: Illustration of variability-aware testing solving the challenges of testing configurable software.

To share also executions for arbitrary values that differ among configurations, a variability-aware interpreter uses conditional values. Furthermore, it is possible to join and share executions efficiently. For example, a configurable program loads a feature selection (e.g., for LETTERS) which can be either true or false. Instead of using only one selection of the feature, a variability-aware interpreter assumes both selections at the same time using conditional values. All family-based analysis tools handle features to have both values. However, the selection of the feature changes the program flow, and thus also affects other values than features (e.g., the String values). Previous tools for variability-aware execution are not able to handle these values efficiently and need to execute the program for these values several times. To handle these values efficiently, a variability-aware interpreter can handle any value as conditional value, depending on the selection of the features.

Current variability-aware interpreters for a WHILE language and PHP show promising results [Kästner et al., 2012b; Nguyen et al., 2014], but both implementations come with major restrictions. The interpreter for WHILE is only a proof of concept and is written for a toy language [Kästner et al., 2012b]. The PHP interpreter Varex is more powerful and is shown to scale for up to 2^{50} configurations. Due to the complicated design of PHP, the interpreter is written to execute one specific application, namely

WorldPress. Varex is a lifted version of an existing interpreter, but only functionality that is actually needed to execute WorldPress is variability-aware. Thus, Varex would require further effort to be applied to other applications than WorldPress and a lifting to support all language features of PHP is too expensive.

Previous work showed that the concept of a variability-aware interpreter scales, but currently only with major restrictions. From previous implementations we learned that lifting of existing interpreters is possible. To be an accepted and powerful tool for testing, the interpreter needs to be able to execute arbitrary applications. Thus, we decided to lift a JVM which interprets Java Bytecode because it has a good specification and a limited amount of instructions. We also learned that it is useful to use a configurable application as help to detect places at the code of the interpreter which need to be lifted. Nevertheless, to use only one application leaves gaps in the interpreter which are not variability-aware. Thus, the interpreter should be applied to several applications to detect functionality that is not lifted.

3.4 Java Virtual Machine

For variability-aware execution, we require an interpreter as basis. Besides Java there are several languages that can be interpreted, such as JavaScript, PHP, and Ruby. To be an accepted and usable tool, the interpreter should be for a widely used language, and the language design should be less complicated than for PHP. We decided to interpret Java Bytecode as basis because of its common specification, its limited amount of instructions and because it is one of the most popular languages.

In this section, we explain the JVM as interpreter for Java Bytecode and point out elements that have to be lifted for variability-aware execution. We focus on details of the JVM that are required for the understanding of a variability-aware Interpreter for Java Bytecode. First, we explain the general architecture of a JVM in Section 3.4.1. Then, we explain how a JVM executes Java Bytecode in Section 3.4.2.

3.4.1 Architecture

Java Bytecode is the compiled form of several programming languages, such as Java, Scala, AspectJ, and with specialized compilers also languages which are usually not compiled to Java Bytecode, such as Python [Juneau et al., 2010]. In contrast to compilation to a system specific machine code, Java Bytecode is platform independent. Nevertheless, to execute this platform-independent bytecode, a platform-dependent machine is required. A JVM is a machine that can execute Java Bytecode independent of the compiled programming language (i.e., a JVM is not specialized to execute programs written in Java). In Figure 3.4, we illustrate the platform and language independent design of JVMs. A specialized compiler transforms a language (e.g., Java or Python) to Java Bytecode. Because the structures of all class files are similar, different JVMs can read and interpret these files, and thus can execute the compiled programs (except

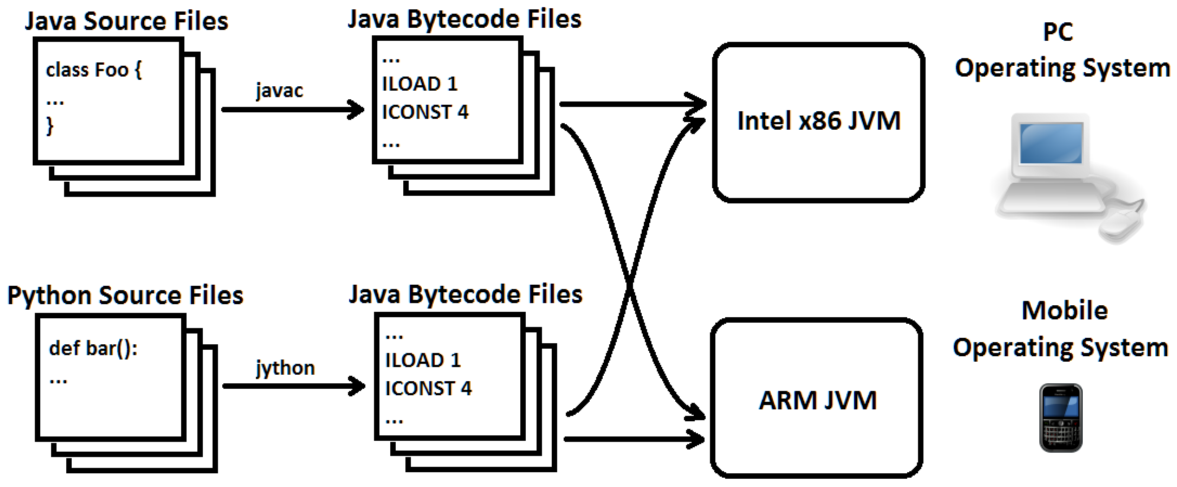

Figure 3.4: Platform and language-independent design of Java Virtual Machines.

of specialized JVMs with a reduced instruction set or an older version than the compiler). To execute the JVM and to communicate with the platform, there are specialized implementations for each operating system (e.g., Windows, Linux, or mobile devices).

All JVMs have a common specification and a common structure. To point out elements of a JVM that have to be lifted for variability-aware execution, we explain the main elements of a JVM. Each JVM consists of a class loader subsystem, runtime data areas, an execution engine, and a native method interface. The class loader subsystem loads the compiled Java Bytecode files into the JVM. The runtime data areas are different types of memory of the JVM, such as a method area to store values of the current methods, a heap that contains objects, Java stacks that store the current method invocations (i.e., the current trace), registers for the program counter (pc) to know the current point of method executions, and native method stacks which are used for native method calls (i.e., functionality to interact with the system usually implemented in C). Furthermore, a JVM has an execution engine that executes the program. Finally, a JVM has a native method interface to interact with system-specific functionality. In Figure 3.5, we summarize and illustrate the main elements of a JVM.

To lift a JVM, several of the parts have to be lifted to handle conditional values. As the class loader is not part of the execution, it does not need to be lifted, and thus can be reused as-is. Because the runtime data areas handle memory of the JVM, the main lifting has to be done there. The method area needs to handle conditional values during executions. The heap might need to point to conditional objects and the pc resisters need to save several points of a method's execution. If still only one method is executed at the same time (i.e., each thread has only one current method), the Java stacks can be reused as-is, else it needs to be lifted, too. To support native method calls, new techniques need to be found, because native implementation usually cannot be lifted (changing native implementations is generally not possible). Finally, the execution engine needs to provide optimized scheduling of method executions to

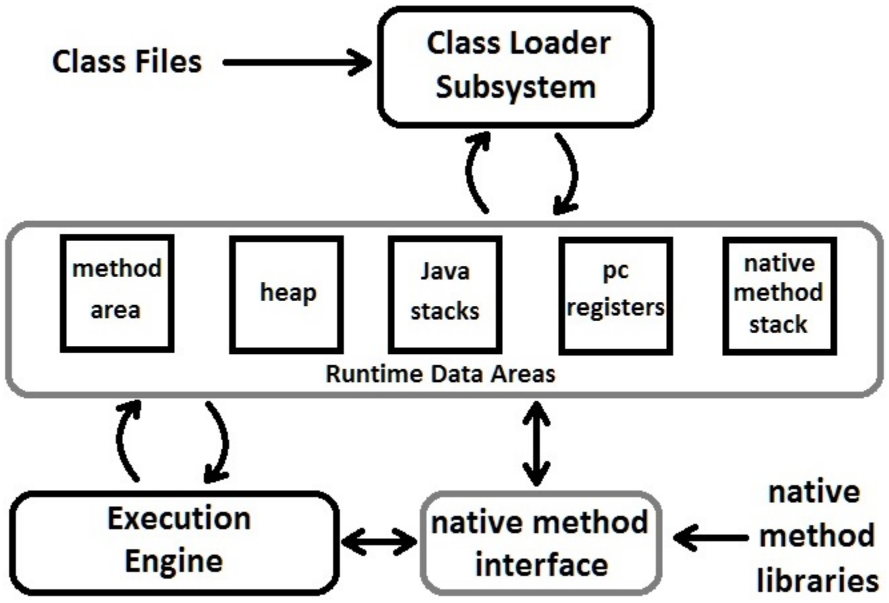

Figure 3.5: Overall architecture and elements of the Java Virtual Machine.

maximize sharing. We cannot provide a general solution for a variability-aware JVM, since the specification of a JVM only defines the general architecture and not how it is implemented. However, most of our realization will share the same challenges. We discuss concrete solutions of these challenges later in Chapter 4.

3.4.2 Execution of Java Bytecode

For variability-aware execution it is necessary to understand how Java Bytecode is executed by a JVM. Therefore, we explain the main elements, the method frame, execution of bytecode instructions, and scheduling in detail.

Method Frame

Each method call creates a new *frame* which is used to store data and partial results of the method's executions [Lindholm et al., 2014]. Each frame consists of an array (i.e., elements are accessed with indexes) for *local variables* and an *operand stack*. The size of local variables and of the operand stack is determined at compile time [Lindholm et al., 2014]. To manipulate the frame, there are several basic instructions: *pop*, *push*, *load*, and *store*. Known from common stack instructions, *pop* returns the top value and *push* adds new value to the operand stack. The instructions *load* and *store* connect the operand stack with the local variables. The instruction *load* gets the value at a given index position of the local variables and pushes it to the operand stack. The opposite instruction *store*, pops a value from the operand stack and saves it at a given index position at the local variables. Furthermore, there are operand stack management instructions which manipulate the values on the operand stack, such as *dup* and *swap*. For example, the instruction *swap* switches the position of the two top values on the operand stack. During execution only one method frame is active. After the method returns, the previous method frame is active again (i.e., method frames are organized as a stack of frames).

Bytecode Instruction

A Java Bytecode instruction is the representation of a single operation similar to an assembler instruction. To interact with the JVM, a bytecode instruction has several abilities, such as modification of the current frame, fields, and invocation of a new frame or returning from the current frame. A Java Bytecode instruction can manipulate the current frame only indirectly using the shown instructions. To explain how a bytecode instruction works, we use the instruction IADD which is used to add two integer values. First, the instruction pops the two top values of the operand stack. Then, it calculates the sum of both values. Finally, the result is pushed back to the operand stack.

Scheduling

To execute instructions in the right order, the order is saved in the method at the compiled class file. To save the current point of the methods execution each frame has an own program counter. After a method is executed, the program counter is usually increased. However, some instructions modify the program counter to execute another instruction than the following (e.g., `for`-loops or `if`-statements). If a method is invoked, a new frame is created, the method parameters are pushed initially to the frame, and the program counter of the new frame points to the first instruction. Furthermore, if a method returns, the top frame is popped, the return value is pushed to the previous frame, and the program counter of the previous frame is increased.

Execution of Java Bytecode

To illustrate the interpretation of Java Bytecode, we use the example in Listing 3.3. The example shows a configurable program that depends on the value of the field `feature`. If `feature` is `true`, the parameter `i` is multiplied by 4, else `i` is divided by 2. Then, `i` is incremented by one. Finally, the method returns the sum of `result` and `i`. The multiplication and the division are executed individually; however, the incrementation and the sum are calculated redundantly for both program variants.

As we are interested how Java Bytecode is executed, we show the compiled program in Listing 3.4. The left side represents the Java Bytecode instructions. `L0` to `L5` are reference points for program counter modifications (e.g., with `GOTO` or `IFEQ`). All shown values are determined during compile time. On the right side, we give a short explanation of each instruction. For example, `result = i * 4` is compiled to `ILOAD 1, ICONST 4, IMUL, ISTORE 2` (Lines 5 to 10). First, the value of `i` is loaded from the local variables and pushed to the operand stack. After this, the constant value 4 is pushed to the operand stack. These two values are then popped from the operand stack, multiplied, and the result is pushed back to the operand stack. Finally, the result is popped from the stack and stored at the local variables at index 2 which represents the value of `result`. To not execute the else-branch, the next instruction `goto` (Line 12) modifies the program counter to point to `L4` (Line 20).

```java
boolean feature = loadConfiguration("feature");

int method(int i) {
    int result;
    if (feature) {
        result = i * 4;
    } else {
        result = i / 2;
    }
    i++;
    return i + result;
}
```

Listing 3.3: Configurable source code example with two program variants, depending on the value of the field `feature`.

```
if (feature):
  L0  ALOAD 0: this       load the reference LV#0 of the method object
      GETFIELD feature    load value of field "feature"
      IFEQ L1             goto L1 if top-value is false
result = i * 4:
  L2  ILOAD 1: i          load LV#1 to operand stack
      ICONST 4            push 4 to operand stack
      IMUL                pop 2 top values,
                          push multiplication to operand stack
      ISTORE 2: result    store top value at LV#2
else:
  L3  GOTO L4             goto L4
result = i / 2:
  L1  ILOAD 1: i          load LV#1 to operand stack
      ICONST 2            push 2 to operand stack
      IDIV                pop 2 top values,
                          push division to operand stack
      ISTORE 3: result    store top value at LV#2
i++:
  L4  IINC 1,1: i         load LV#1, increment value,
                          store value to LV#1
return i + result:
  L5  ILOAD 1: i          push LV#1 to operand stack
      ILOAD 2: result     push LV#2 to operand stack
      IADD                pop 2 top values,
                          push sum to operand stack
      IRETURN             pop current method frame,
                          push last top value to operand stack
```

Listing 3.4: Compiled Java Bytecode for the configurable example in Listing 3.3. The left side shows the compiled bytecode instructions. On the right side we show a short explanation of the corresponding instructions.

3.5 Variability-Aware Execution of Java Bytecode

To realize variability-aware execution for Java Bytecode, we first discuss the ideas of *shared data* and *shared execution* [Nguyen et al., 2014], and how they can be applied to a JVM. Then, we discuss how Java Bytecode is executed variability-aware using the example of Listing 3.3 and Listing 3.4.

Shared Data

During a program's execution values differ among configurations. To represent these values specific for the corresponding configurations we use conditional values. Because values are equivalent for several configurations, the conditional values can be simplified to share data (e.g., Choice(Feature, 1, 1) can be simplified to 1).

To implement shared data in a JVM, the *frame* needs to be able to handle conditional instead of concrete values. Thus, the frame needs to store conditional values and the instructions, such as push and pop, need to be adjusted. Furthermore, *fields* that can store only one value need to store a conditional value. Finally, a *reference* can point to different objects for different configurations. Thus, references and all functionality that accesses these references have to be lifted, too.

Shared Execution

To reduce the effort for calculations, equivalent executions among configurations are shared [Austin and Flanagan, 2012; Kim et al., 2012; Kästner et al., 2012b; Nguyen et al., 2014]. For a variability-aware interpreter each instruction is executed within a specific configurations space. Instructions that are executed with the context *True* are shared among all configurations. When an instruction is executed with a specific context, such as $A \wedge \neg B$, the instruction is shared among all configurations where the context holds. The goal of variability-aware execution is to minimize the number of executed instructions by sharing instructions among as many configurations as possible.

The selection of features can change the program flow. Thus, the scheduler needs to decide which instruction has to be executed next. Therefore, the program counter needs to point to the instruction that would be executed specific for the configurations spaces. Because the program counter can point do several instructions of a method, also the program counter needs to store a conditional value. To determine which of these instructions should be executed next, it is usually sufficient to execute the instruction that is most behind in the method's execution. There are special cases, such as return statements, that have to be handled differently. We give a detailed discussion of scheduling later in Section 4.2.4.

To explain the details of variability-aware execution of Java Bytecode, we reuse the example of Listing 3.3 and Listing 3.4. In Table 3.1, we illustrate the corresponding executions for both values of the field feature, `true` and `false`. Furthermore, we show the trace of variability-aware execution that executes the program for both values simultaneously. To introduce both selections of the field `feature`, the method `loadConfiguration` directly stores the value `Choice(F, true, false)`. The example represents the execution for the parameter `i = 6`. The method has three local variables, #0 is the object reference (i.e., `this`), #1 is `i`, and #2 is `result`. Furthermore, the method has a stack of a maximal size of 2. Both, the local variables and the size of the stack are determined during compilation [Lindholm et al., 2014]. All traces are initialized similarly during method invocation. The first local variable is always a reference value to the object of the method [Lindholm et al., 2014] (this does not hold for static calls). The following local variables are the method parameters. Thus, the local variable with index 1 (`i`) is initialized with 6. In the following instructions we do not show the values for index 0 because the reference to the object does not change. All changes by instructions are highlighted bold. The gray background indicates the method's execution skips some instructions in case of a `goto` or `if`.

The left half represents traces for default product-based executions, where the field feature can only have one concrete value. We see that many (8 of 17) instructions are done redundantly. Thus, the overhead for execution is already high for this simple example. On the right side, we show the corresponding trace for variability-aware execution. The value of the field `feature` is a choice which can be `true` and `false` depending on the selection of `F`. Instead of one concrete value the variability-aware interpreter uses conditional values for local variables and values in the operand stack. We see that all instructions have to be executed only once. Thus, the overhead for redundant calculation can be reduced. However, the effort for calculating with conditional values is increased compared to default executions. In contrast to other variability-aware approaches, the variability-aware interpreter can join the executions after the if-else branches. Additional, the shared instructions IINC 1,1, ILOAD 1, and ILOAD 2 do not require additional effort for calculations with conditional values. The effort is even equivalent to a default execution. Only for IADD the effort is higher because we need to calculate a cross product of both values `i` and `result`. Finally, the top value is the value which is returned. Instead of returning one value, the variability-aware interpreter returns both values as a choice.

We discussed the main principles of execution of Java Bytecode. Furthermore, we discussed the general procedure of a variability-aware interpreter. We do not present implementation details here, because a JVM is an abstract machine and all implementations differ. In Chapter 4, we discuss our concrete implementations of a variability-aware interpreter in detail. Because the overall structures of all JVMs are similar, they share the same difficulties and challenges for lifting and efficient variability-aware execution. Thus, our solutions can be adopted to develop other variability-aware JVMs and variability-aware interpreters in general.

	product-based				variability-aware	
	feature = true		feature = false		feature = Λ(F, true, false)	
Instruction	Local	Stack	Local	Stack	Local	Stack
initial state	**this** **i = 6** 0		**this** **i = 6** 0		**this** **i = 6** 0	
ALOAD 0	6 0	**this**	6 0	**this**	6 0	**this**
GETFIELD	6 0	**true**	6 0	**false**	6 0	**Λ(F, true, false)**
IFEQ L1	6 0		6 0		6 0	
ILOAD 1	6 0	**6**	▨	▨	6 0	**Λ(F, 6, ⊥)**

ICONST 4	6
	0
		6
		4
			▨	▨	6
					0
						Λ(F, 6, ⊥)
						Λ(F, 4, ⊥)

IMUL	6 0	**20**	▨	▨	6 0	**Λ(F, 20, ⊥)**
ISTORE 2	6 **20**		▨	▨	6 **Λ(F, 20, 0)**	
GOTO L4	6 20		▨	▨	6 Λ(F, 20, 0)	
ILOAD 1	▨	▨	6 0	**6**	6 Λ(F, 20, 0)	**Λ(F, ⊥, 6)**

ICONST 2	▨	▨	6
			0
				6
				2
					6
					Λ(F, 20, 0)
						Λ(F, ⊥, 6)
						Λ(F, ⊥, 2)

IDIV	▨	▨	6 0	**3**	6 Λ(F, 20, 0)	**Λ(F, ⊥, 3)**
ISTORE 2	▨	▨	6 **3**		6 **Λ(F, 20, 3)**	
IINC 1, 1	**7** 20		**7** 3		**7** Λ(F, 20, 3)	
ILOAD 1	7 20	**7**	7 3	**7**	7 Λ(F, 20, 3)	**7**
ILOAD 2	7 20	7 **20**	7 3	7 **3**	7 Λ(F, 20, 3)	7 **Λ(F, 20, 3)**
IADD	27 20	**27**	7 3	**10**	7 Λ(F, 20, 3)	**Λ(F, 27, 10)**

Table 3.1: Traces and method frame states for the executions of bytecode instructions of the configurable example in Listing 3.3 and Listing 3.4. We use Λ as abbreviation for a Choice. ⊥ indicates an unknown or unnecessary value. The left side represents the product-based executions for feature = true and for feature = false. The right side shows the trace and values for variability-aware execution using conditional values.

3.6 Summary

In this chapter, we introduced into the challenges with testing of configurable software. We discussed the pitfalls of product-based analysis, such as redundant calculations and the scalability problem. With family-based analysis, we introduced into a strategy to solve these challenges, to provide efficient analysis for configurable software. We discussed the general principles of variability-aware execution and the concepts of a variability-aware interpreter. We introduced into the general principles and architecture of a JVM which we want to use for variability-aware execution. Finally, we discussed the principles of variability-aware execution of Java Bytecode.

4. Implementation: Variability-Aware Execution with VarexJ

In this chapter, we discuss our realization of a variability-aware JVM. At first, we discuss our decision for a JVM that suits our requirements in Section 4.1, namely JPF. Furthermore, we discuss the general architecture of JPF to point out places that have to be lifted for variability-aware execution. Based on JPF and the concepts of the previous chapter, we present basic implementation details of our variability-aware interpreter VarexJ in Section 4.2. We discuss specialized extensions of VarexJ in Section 4.3. In Section 4.4, we discuss further optimizations and improvements of variability-aware execution.

4.1 Java Pathfinder

In this section, we discuss why we choose to lift JPF for variability-aware execution. The goal of this work is to provide a variability-aware interpreter for Java. Therefore, it would be possible to write a new interpreter specialized on variability-aware execution. However, to do this is an expensive and error-prone task. Furthermore, we want to show how existing interpreters can be lifted, that the approach is feasible, and that the task is realizable in a limited amount of time.

There are several open-source JVM implementations. First, we first discuss our requirements on a JVM for a prototypical but efficient implementation of a variability-aware interpreter. Second, we discuss our decision for the JVM JPF. Third, we explain the overall structure and architecture of JPF, for understanding of the interpreter and to point out elements that have to be lifted.

4.1.1 Requirements and Discussion

To select the most appropriate JVM, we state several requirements. The first requirement is *simplicity*, because the time for this project is limited. A JVM is a complex and complicated program. To modify such a program requires a basic understanding of it. If the JVM is too complicate, has too many optimization mechanisms, or is not well structured, a variability-aware lifting is not possible within our project. For a suiting JVM, it should be easy to identify basic parts of the JVM's implementations, such as execution of bytecode instructions, the scheduling mechanism, and the representation of method stacks.

All JVMs suit different purposes with different implementations. As we want to change how Java Bytecode instructions are interpreted, it is necessary that the JVM has an *interpreter*. Java Bytecode does not necessarily need to be interpreted, some JVMs, such as Maxine [Wimmer et al., 2013] and Jikes [Alpern et al., 2005], compile the Java Bytecode directly into machine code. Execution of compiled machine code might be more efficient than interpreting Java Bytecode, however, to introduce variability into compiled machine code is also more complicated and error prone than to adjust the executions of an interpreted instruction. In particular, for an interpreter we can lift internally used values of the executed program, what seems to be infeasible for machine code.

The library FeatureExprLib[1] of the TypeChef [Liebig et al., 2013; Kästner et al., 2012a, 2011] project eases the use and reasoning about feature expressions. To reuse this existing infrastructure written in Scala, the interpreter should be *implemented in Java* or other languages compiled to Java Bytecode. For reasoning on feature expressions, the project allows to use both, SAT solvers and BDDs. Furthermore, the library supports loading feature models saved as dimacs file. FeatureExprLib has already been used in previous variability aware-interpreters [Nguyen et al., 2014; Kästner et al., 2012b].

Lifting an interpreter requires major changes on main parts of the interpreter. To change a, for the most parts unknown, software system is error prone. To ensure that the interpreter works after changes have been performed an *extensive test suite* is required. Because even minimal changes can cause unpredictable faults, a test suit increases the confidence of the programmer when changing the interpreter. The test suite can only test unconditional inputs and values, however, these test cases also have to work. As also executions with conditional values require specialized tests, existing test have to be adjusted or new test have to be implemented.

We summarize our requirements on a JVM that we want to extend as follows:

- Simplicity to finish the project in limited time,
- Interpreter for Java Bytecode,
- Written in Java to reuse existing libraries,
- Extensive test suite to detect faults early.

[1]https://github.com/ckaestne/TypeChef/tree/master/FeatureExprLib

We discussed several requirements on a JVM for an efficient lifting. These requirements reduce the number of possible JVMs which we extend for a variability-aware execution. The reduction of possible JVMs comes with several pitfalls; however, they are not necessarily negative or important, because our goal is only a prototypical implementation. Because of the requirements of simplicity, interpretation of Java Bytecode, and written in Java, the JVM might be slow compared to other JVMs and limited (e.g., not all programs can be executed). The requirements increase the cost-benefit ratio, to achieve the goals of this work within a limited amount of time.

Selecting a Suitable Virtual Machine

There are several open-source JVMs collected in an article on wikipedia.[2] We tried out two JVMs, namely Maxine [Wimmer et al., 2013] and Jikes [Alpern et al., 2005], which are written in Java and still used for research purposes. However, we found out that both JVMs compile Java Bytecode instead of interpreting it. Thus, they did not suit our requirements. We did not found other suiting JVMs for our requirements in this article, especially, because of the restriction to Java and interpreting of Java Bytecode. The article might contain some more interpreters which would suite our purposes; however they are not supported and developed anymore. Thus, we decided to use the interpreter of JPF, which is not listed in the article. JPF is a software model checker that uses Java Bytecode instructions as transitions between states [Visser et al., 2003]. Previous work on variability-aware execution [Kim et al., 2012; Kästner et al., 2012b] also used JPF. However, they used the model checking abilities of JPF to split states and to execute separate paths. When the model checking abilities are disabled, JPF behaves like an interpreter for Java Bytecode. In this work, we also extend JPF for variability-aware execution, but with a different approach. Furthermore, we already have experience with JPF in the context of variability-aware execution from our previous work [Meinicke, 2013; Thüm et al., 2014c].

4.1.2 Architecture of Java Pathfinder

We briefly discuss the overall architecture of JPF and its dependencies to the host JVM and the application to test, to point out places which have to be lifted for variability-aware execution. JPF is a Java program itself which requires a host JVM to run. Thus, JPF interprets the Java Bytecode program which should be tested, the host JVM executes JPF, and the operating system runs the host JVM. In Figure 4.1, we illustrate the dependencies of the different layers. The host JVM is a platform-specific installation, which provides a native library. The native library contains implementations, usually written in C, to interact with the operating system (e.g., to access the file system). Furthermore, the host JVM provides a shared library (rt.jar) containing standard implementations, such as String or List. This library can be used by any Java application, such as JPF and the application which should be executed. JPF is executed by a host JVM and interprets the compiled Java Bytecode of the application to test. To simply

[2]http://en.wikipedia.org/wiki/List_of_Java_virtual_machines

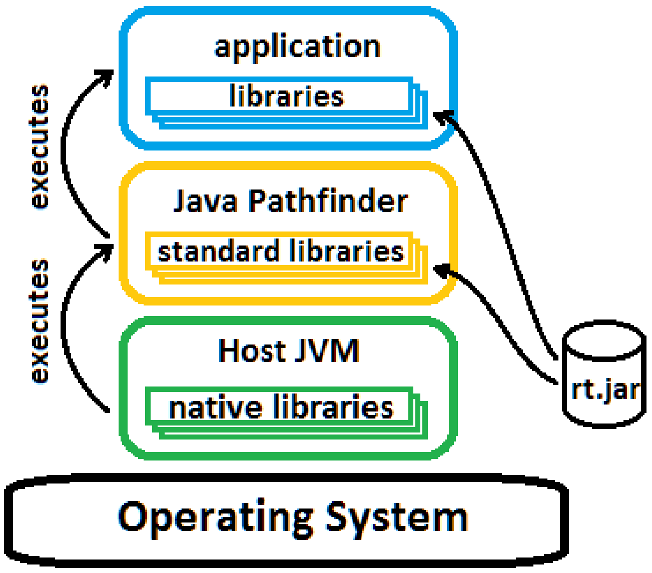

Figure 4.1: Interaction of Java Pathfinder with the host Java Virtual Machine, the operation system, the application to execute, and the used libraries.

run the application the complete layer of JPF is unnecessary. As we want to adjust the way how the application is interpreted and how values are represented, we require the additional layer of JPF.

JPF is specialized for model checking, and thus contains functionality which is not necessary for our purposes (e.g., backtracking or different graph-search strategies). To explain the architecture of JPF, we only show the parts that are necessary for interpretation of Java Bytecode. In Figure 4.2, we illustrate the top-level structure of JPF containing basic elements for interpretation of Java Bytecode which have to be lifted. First, JPF parses the compiled Java Bytecode of the application which should be executed ((1) in Figure 4.2). These class files are transformed into internal representations `ClassInfo`, `MethodInfo`, and `Field`, containing information about methods, fields, and types. The main element for interpretation is `ThreadInfo`. It contains the method frames organized as heap (2), and runs the main loop executing Java Bytecode instructions (3). The `StackFrame` represents the current methods execution. It saves the current state of execution (`ProgramCounter`) and the current data for local variables and operands in an array (`frame`). In the main loop the bytecode instructions are executed (4). Bytecode instructions can interact with the current method frame (5), can push or pop a frame to the method heap (6), and can interact with objects represented by `ElementInfo` to set or get the value of fields (7). Furthermore, JPF has a mechanism to redirect native method calls to the host JVM. We discuss details on native method calls later, because they require a specialized handling.

For variability-aware execution of Java Bytecode instructions several parts of JPF's interpretation and data representation have to be lifted. The `StackFrame` requires an efficient encoding of conditional values, because it is a central element which represents

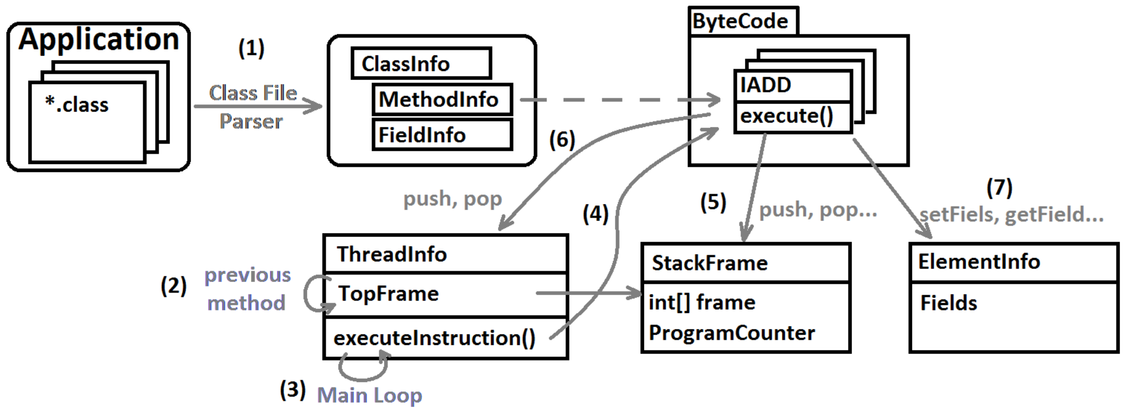

Figure 4.2: Top-level structure of Java Pathfinder, containing basic elements which need to be lifted.

the data of the method executions. To execute instructions conditional and to handle conditional values of the frame, all instructions have to be lifted. There are currently 183 instructions supported by JPF. As several instructions are very similar because they only differ in the type of the supported value (e.g., `IADD` for addition of int values and `DADD` for double). Furthermore, the values of fields need to be conditional, too. Because some instructions are only executed in specific contexts, the main loop needs to handle conditional execution paths, while the sharing of executed instructions is optimized. Several parts of JPF need to be lifted. However, most parts can stay unchanged, because they either are only required for model checking (e.g., search algorithms), or they do not need to handle variability, such as parsing of bytecode.

In this section, we discussed our requirements on a JVM. We showed that JPF suits best for our purposes and requirements. We discussed the top-level structure of JPF and pointed out the major elements which have to be lifted.

4.2 Fundamental Extensions for Variability-Aware Execution

In the previous section, we explained the overall architecture of JPF. In this chapter, we discuss our realization of a variability-aware interpreter called VarexJ, based on JPF. First, we present our approach of lifting and refactoring. In this thesis, we use the term refactoring in the way that we extend the virtual machine to handle conditional values, while we preserve its overall behavior. Then we discuss our variability-aware extensions of the main parts of the JVM explained in Chapter 3, the method frame, execution of bytecode instructions, and scheduling. VarexJ is publicly available online at GitHub.[3] The repository contains all sources required to execute the variability-aware interpreter. Furthermore, we give a guide explaining how to install and execute VarexJ including all provided program parameters and configuration options.

[3]https://github.com/meinicke/VarexJ

4.2.1 Incremental Lifting of Java Applications

Lifting of a JVM is a complex and challenging project. In this subsection, we discuss our general strategy for lifting of JPF. We share our practices and experiences that we gained during this project. Our recommendations can be generally applied for lifting of application, because lifting can also be applied to other programs than JVMs. Furthermore, our experiences may useful for software development in general, especially when large refactorings are required.

Refactoring

The overall goal of lifting a program is that all necessary methods and data structures can handle conditional values. To achieve this goal is only incrementally feasible. We experienced the following refactorings that are required for lifting:

- Changing types (e.g., from `int` to `Conditional<Integer>`) to store conditional values in:
  - Parameters
  - Fields
  - Local variables
  - Return values
- Adding of a context as additional parameter, to perform context sensitive changes.
- Not variability-aware methods need to be lifted with map functions.

To describe the challenges of lifting, we use the simple increment function in Listing 4.1. The method `increment` gets a delta as input and applies it to a field that saves intermediate values. After the delta is applied, the new value is returned.

```
int current = 0;

int increment(int delta) {
    current = current + delta;
    return current;
}
```

Listing 4.1: Example to illustrate the effort of refactorings for lifting of a simple method (e.g., when `delta` should be a conditional value).

The example is simple, it does not contain any objects or method called and has only minimal lines of code. Nevertheless, the example shows the challenges for lifting, because it already requires five refactorings (highlighted with gray and adding of the parameter for the context) to be completely variability-aware. All the types need to be changed from `int` to `Conditional<Integer>`, which already requires three refactorings. Furthermore, the part which increments the field (`current + delta`) needs to be refactored with fmap. Finally, to only increment the value of `current` for specific feature selections, an additional parameter for the context needs to be added. In Listing 4.2, we show the result after all five refactorings are applied.

```
Conditional<Integer> current = new One<>(0);

Conditional<Integer> increment(FeatureExpr ctx,
                               Conditional<Integer> delta) {
    current = current.fmap((c) -> {
        return delta.fmap((d) -> {
            return new Choice<>(ctx, new One<>(c + d), new One<>(c));
        });
    }).simplify();
    return field;
}
```

Listing 4.2: Refactored example of Listing 4.1 supporting conditional values.

To refactor a method completely, requires high effort and is error-prone, because the refactoring of one method usually requires further refactorings of other methods. For example, if the field `current` is used by other methods than `increment`, these methods would need to be refactored, too. To test the virtual machine during development, we require a type save and compilable state. Thus, we introduce the method `getValue` which transforms a conditional value to a concrete value (e.g., One(1) is transformed to 1). In Listing 4.3, we show the implementation of `getValue` for conditional values. The implementation only returns the concrete value if the conditional is of type `One`. If the conditional value is a `Choice`, an exception is thrown, because if a `Choice` should be transformed to a concrete value only one entry could be returned and information would get lost.

In Listing 4.4, we illustrate the benefits of a transformation from conditional to concrete values. A typical change during lifting is that a parameter of a method can be conditional. When changing the type of parameter `delta`, a complete refactoring would be necessary to result in a type save and compilable state. To reduce this effort temporarily, `delta` is transformed to a concrete value. The call of `getValue` is only valid if the method `increment` is called with a value of type `One`. As long as the method is not called with a `Choice`, the effort for further refactorings can be saved. When the method is called with a `Choice`, this is signaled with a runtime exception, which prints the current stack trace and the value of the choice to the console. At this point, the method `increment` requires further refactorings. To use `getValue` is especially useful when a refactoring requires that several methods need to be changed (e.g., when a field is changed to conditional), thus a complete refactoring of all places is too expensive. We experienced that the refactoring of single fields or methods can lead to hundreds of places which need to be adjusted and that refactoring without the method `getValue` gets infeasible.

```java
interface Conditional<T> {
    T getValue();
}
class One<T> implements Conditional<T> {
    T value;
    T getValue() {
        return value;
    }
}
class Choice<T> implements Conditional<T> {
    T getValue() {
        throw new RuntimeException("getValue on Choice called:"
                                   + toString());
    }
}
```

Listing 4.3: Transformation from conditional to concrete values for incremental lifting using runtime exceptions.

```java
int current = 0;

int increment( Conditional<Integer> delta) {
    current = current + delta.getValue() ;
    return current;
}
```

Listing 4.4: Example for incremental lifting of the example in Listing 4.1 to temporary avoid a complete refactoring.

Refactoring: New Parameter

We showed that lifting requires several refactorings. Modern development environments, such as Eclipse[4] and IntelliJIDEA,[5] reduce the effort with automated refactorings. Refactorings allow changing method signatures, such as changing return types, changing parameter types or adding of new parameters. One of the most important refactorings for a variability-aware lifting is to add a new parameter for the context. When adding a new parameter to a method, the parameter needs to be added to all calls of this method. For the context we discovered three possible values which can be used as initial value shown in Listing 4.5: a null value (`method1`), the context True (i.e., a tautology) (`method2`), or the name of the contexts (we always use ”ctx” as name for the context parameter) (`method3`). When using True as initial value, this value is type-save, but it is not necessarily the right value. To detect faults caused by simply using True requires high effort, because it is not always obvious how a fault is caused. The second variant of using the null value temporary creates a type save state, however, at the point when the context is required, the program throws a null-pointer

[4]https://www.eclipse.org
[5]http://www.jetbrains.com/idea

exception (in the best case) or it ignores the context which leads to false computations. The safest solution is to use the context defined as parameter of the calling method. Because the calling method might not have a parameter, too, also this method needs to be refactored. The third solution requires high effort because the refactorings have to be applied until a context is found. However, it is correct and does not introduce defects because the parameter is just added and passed on to further methods which require it. We recommend using the third way, even if it requires more initial effort than the other variants. There is always a tradeoff between initial effort and effort for debugging of failures caused by the variants.

```
method1() { increment(null, 1); }
method2() { increment(True, 1); }
method3(FeatureExpr ctx) { increment(ctx, 1); }
int increment( FeatureExpr ctx , Integer delta) { ... }
```

Listing 4.5: Alternative refactorings to introduce a new parameter for the context.

Revision Control System

During development, we tried to introduce as small changes as possible (i.e., only minimal parts are lifted at once). Though, sometimes a refactoring of a method signature or a change of a field type requires hundreds of changes. After all required changes are applied to reach a type-save state, it is necessary to check that the system still works as desired. To check this, an extensive test suite is required, and needs to be applied. Whenever a test fails, the corresponding fault needs to be found and debugged what might require high effort. This effort can be reduced when only minimal changes are applied, and thus the comparison with the previous version, which can lead to the possible fault, is simpler. Anyhow, sometimes the defect cannot be found. In this case, it is necessary to revert the changes on the system to a previous version. To reduce the amount of lost changes and effort, it is useful to commit even minimal changes to a revision control system.

Continuous Integration

The goal of this work is an efficient testing and execution of configurable Java applications. During development, we checked the progress of the JVM on smaller programs, as shown in our examples, to check certain properties, such as return of conditional values, conditional fields, or conditional lists. Furthermore, we also executed larger configurable programs with up to 10k lines of code [Kim et al., 2012; Apel et al., 2013d]. To keep track of the performance of the executions, we used the continuous integration system Jenkins.[6] With continuous integration, builds and tests can be applied automatically on a server. Furthermore, the system can keep track of test cases, their success, and execution times. With such statistics eventually negative trends in performance can be detected early.

[6] http://jenkins-ci.org/

We summarize our experiences and recommendations for lifting of applications as:

- Variability should not be introduced at every place at once.
- Run-time exceptions reduce initial effort and can be necessary for a compilable state, but should be used with caution.
- New method parameters should not be initialized with a null or default values. A complete refactoring should for this parameter should be instead.
- Even minimal changes should be tested.
- A revision control system should be used to reset the system to a valid state.
- Commit small changes to a revision control system to minimize lost effort.
- Continuous integration helps to keep track of test performances over time.

We discussed how to incrementally lift large applications, such as a JVM. We discussed our general procedure with minimal changes to the source code, runtime exceptions, exhaustive testing, revision control, and continuous integration. In the next sections, we use this procedure and discuss implementation details for parts of the JVM discussed in Chapter 3, the method frame, execution of bytecode instructions, and scheduling.

4.2.2 Method Frame

A *frame* is used to store data and partial results of a method's execution [Lindholm et al., 2014]. Each frame consists of an array for local variables and an operand stack. In JPF, both, the local variables and the operand stack are implemented as one array. The first slots of the array are reserved for local variables, depending on the amount of necessary local variables. Local variables can be accessed directly with an index. The other part of the array represents the operand stack. The top position of the stack is saved in an additional value. To know whether a value is a reference or not, JPF uses a bitset. There are several basic instructions for manipulation of the stack frame, such as *pop*, *push*, *load*, and *store*, as well as further instructions which manipulate the operand stack, such as *dup*, and *swap*. In Listing 4.6, we show the default stack frame implementation of JPF. It contains the three fields `slots`, `top`, and `isRef` to store local variables and operands, as well as the methods for modifications `push`, `pop`, `store`, and `load`.

The method frame is a central part of the execution of a method. Thus, the method frame has to be implemented memory and time efficient. A variability-aware stack frame is a main challenge of lifting a JVM, because it is necessary to find the optimal balance between sharing and cross products with fmap. For a variability-aware frame, an initial approach is to keep the structure of the frame and to use `Conditional<StackFrame>` objects instead, as shown in Listing 4.7. This approach might be simple, and the program only needs to be adjusted at the parts that are calling the stack frame. However, there is only minimal sharing among stack frames of the same method, and every time an instruction is performed on the stack frame, the stack frame might need to be cloned (see Line 5).

```
class StackFrame {
    protected int[] slots;
    protected int top;
    protected BitSet isRef;
    StackFrame(int nLocals, int nOperands) {
        slots = new int[nLocals + nOperands];
        top = nLocals − 1;
        isRef = new BitSet(nLocals + nOperands);
    }
    void push(int newValue) { slots[++top] = newValue; }
    void pop() { return slots[top−−]; }
    void store(int index) { slots[index] = pop(); }
    void load(int index) { push(slots[index]); }
}
```

Listing 4.6: Simplified method frame implementation of Java Pathfinder.

```
$ StackFrame sf = new StackFrame(2, 10);
$ sf.push(10);
Conditional<StackFrame> sf = new One<>(new StackFrame(2, 10));
sf.fmap((frame) −> {
    StackFrame clone = frame.clone();
    frame.push(10);
    return new (ctx, new One(frame), new One(frame))
}
```

Listing 4.7: Inefficient implementation for a variability-aware method frame.

To create a more efficient variability-aware stack frame, we started with changing the type of `slots` from `int[]` to `Conditional<Integer>[]` (`Conditional<int[]>` is also possible, but has less sharing). All elements at the same index position in `slots` can be shared, such that it is a memory saving representation of the stack. When the stack is conditional, also the top positions of the stack can be different, thus also `top` has to be a conditional value, as well as the reference map `isRef`. For basic instructions, push, pop, load and store, the representation is sufficient and efficient. However, when applying an operand stack manipulation instruction, such as swap (shown in Listing 4.8), the instruction requires a cross product over all three elements: `slots`, `top`, and `isRef`. This extensive cross product can get expensive, and thus changing `slots` to `Conditional<Integer>[]` is not sufficient for our purposes.

For an efficient representation that supports sharing and does not require extensive cross products, we decided to implement a new stack frame. To get rid of conditional top and reference values, we decided to use conditional operand stacks as `Conditional<Stack>`. The representation does not support sharing among different operand stacks. For support of sharing, we decided to define the local variables as `Conditional<Entry[]>`, because the index of local variables is not conditional. We provide flexible and changeable implementation with a factory using the

```java
public void swap () {
    int t = top−1;
    int v = slots[top];
    boolean isTopRef = isRef.get(top);
    slots[top] = slots[t];
    isRef.set( top, isRef.get(t));
    slots[t] = v;
    isRef.set( t, isTopRef);
}
```

Listing 4.8: Java Pathfinder implementation of the operand stack management instruction `swap`.

`IStackHandler` interface (currently with only one variant). In Listing 4.9, we show an excerpt of the class `StackHandler` which replaces the slots of `StackFrame`. Because `top` and `isRef` are no longer conditional, functions, such as swap, do not require expensive cross products with `fmap` over these elements. The implementation might be similar to the first approach of `Conditional<StackFrame>`, but it increases sharing for local variables, reduces effort for cloning, and thus reduces memory consumption and increases efficiency. We changed the signatures of all instructions (e.g., from `push(int)` to `push(FeatureExpr ctx, Conditional<integer>)`), because operand stack management instructions should be applied only in a specific context. The change to the `StackHandler` implementation caused a high speed-up compared to directly lifting the original implementation of `StackFrame`.

4.2.3 Bytecode Instructions

With the class `StackHandler`, we have an efficient representation for the method frame. For the interpretation of Java Bytecode instructions, JPF has a class for each instruction, which provides a method `execute` that can modify the method frame, the program flow, and fields. The instructions are called from the main loop of `ThreadInfo`. Because the main loop does not need to know what instruction is executed, all instructions provide a common signature shown in Listing 4.10. The instruction is called with the method `execute`. To access the current method frame, to create or to return from methods, and to access fields, the method `execute` gets a reference to `ThreadInfo` as parameter. The instruction can then modify the frame or the fields directly. Finally, the method `execute` returns the next instruction. This returned instruction is then used as new program counter of the current method.

For complete support of variability-aware execution, all instructions have to be lifted. First, the instructions need an additional parameter for the context to execute the instruction only for a specific context. Second, all calls of the `StackFrame` have to be changed to support conditional values. Third, the return type needs to be changed to `Conditional<Instruction>`, because the instruction might return different pointers to the next instructions (e.g., within an if-instruction). The required refactorings for lifting of instructions are shown in Listing 4.11 and highlighted with gray.

```java
public class StackHandler implements IStackHandler {
    private Conditional<Entry>[] locals;
    public Conditional<Stack> stack;
    public StackHandler(int nLocals, int nOperands) {
        locals = new Conditional[nLocals];
        stack = new One<>(new Stack(nOperands));
    }
    public void push(FeatureExpr ctx, Conditional<Integer>) {
        stack.fmap(...) } // apply stack functions
}
public class Stack {
    public int top = −1;
    public Entry[] slots;
    public Stack(int nOperands) {
        slots = new Entry[nOperands];
    }
    public void swap() {
        Entry A = slots[top − 1];
        Entry B = slots[top];
        slots[top − 1] = B;
        slots[top] = A;
    }
}
public class Entry {
    boolean isRef;
    int value;
}
```

Listing 4.9: Efficient variability-aware stack frame implementation used by VarexJ.

```java
public Instruction execute(ThreadInfo ti) {
    StackFrame frame = ti.getTopFrame();
    ... do something e.g.:
        int value = frame.pop();
    ...
    return getNext(ti);
}
```

Listing 4.10: Signature of Java Bytecode instructions in Java Pathfinder.

```java
public Conditional<Instruction> execute(FeatureExpr ctx,ThreadInfo ti){
    StackFrame frame = ti.getTopFrame();
    ... do something e.g.:
        Conditional<Integer> value = frame.pop(ctx);
    ...
    return getNext(ctx, ti);
}
```

Listing 4.11: Signature of variability-aware Java Bytecode instructions in VarexJ.

```java
class IADD  extends JVMInstruction {
    public Instruction execute (ThreadInfo ti) {
        StackFrame frame = ti.getTopFrame();
        int v1 = frame.pop();
        int v2 = frame.pop();
        int sum = v1 + v2;
        frame.push(sum);
        return getNext(ti);
    }
}
```

Listing 4.12: Example bytecode instruction IADD in Java Pathfinder.

```java
public class IADD extends JVMInstruction {
    public Conditional<Instruction> execute(
                                FeatureExpr ctx, ThreadInfo ti) {
        StackFrame frame = ti.getTopFrame();
        Conditional<Integer> v1 = frame.pop(ctx);
        Conditional<Integer> v2 = frame.pop(ctx);
        Conditional<Integer> sum = v1.fmap((x1) -> {
            return v2.map((x2) -> { return x1 + x2; });
        }).simplify();
        frame.push(ctx, sum);
        return getNext(ctx, ti);
    }
}
```

Listing 4.13: Variability-aware implementation of IADD in VarexJ.

To discuss details of lifting bytecode instructions, we use the class for the instruction IADD shown in Listing 4.12. First, the instruction gets the frame of the current method, which is equivalent to the top frame. Then, the two values which should be added, `v1` and `v2`, are popped from the frame. After this, the sum of both values is calculated and pushed back to the frame. Finally, the next bytecode instruction that should be executed is returned.

In Listing 4.13, we show the result of the variability-aware lifting of `IADD`. Again the two values which should be added are popped from the frame, however, only for the specified context. These values are then added using a cross product resulting in the conditional value `sum`. The sum is then pushed to frame for the specified context. At the end, the method returns the next instruction that is specific for the context.

We illustrated the general procedure for lifting of bytecode instructions. This lifting has to be applied to all 183 instructions of JPF for a complete variability-aware instruction set. Because of the high number of instructions, we did not lift all instructions initially. We started with integer values only and used runtime exceptions for those instructions that are not lifted. After the lifting for integers is shown to work, we applied the refactorings to all other instructions step by step. Finally, we were able to lift all bytecode instructions to provide a completely variability-aware instruction set.

4.2.4 Scheduling

The task of a variability-aware scheduler is to determine which instruction has to be executed next to maximize sharing, and thus to reduce the effort for program execution. We already discussed a basic strategy for scheduling in Section 3.5. In this section, we discuss further details on variability-aware scheduling and of scheduling in VarexJ.

With a conditional frame and conditional execution of instructions, the basics for variability-aware execution are given. In a class file, the order of bytecode instructions for each method is defined. There are five cases how to determine which instruction is executed next: (1) the instruction which follows directly in the list of instruction, (2) an instruction points to the next instruction which does not follow directly (e.g., an if-else branch or goto), (3) a new method is called, (4) the current method returns and the calling method is continued, (5) the executed instruction causes an exception (e.g., division by null), or is an exception itself. The default scheduling mechanism is oblivious of these types of instructions and simply executes the instruction which is returned by the previous instruction (see Listing 4.14). However, for variability-aware scheduling, these types of instructions need to be handled differently.

```
public void schedule() {
    Instruction pc = initialPC();
    while (pc != null) {
        pc = pc.execute();
    }
}
```

Listing 4.14: Simplified default scheduling mechanism of Java Pathfinder.

Not all method executions have several paths for different contexts and the original scheduling mechanism is often sufficient. Sometimes the executions paths are split (e.g., caused by an if-statement on a conditional value). Thus, the current program counter is a conditional value that can point to several instructions at the same method. To support joining (i.e., different execution paths are executed together again) and to maximize sharing it is necessary to determine which of these instructions should be executed next, because only one instruction can be executed at once. As instructions are saved as ordered list in the methods descriptions (i.e., in `MethodInfo`), also the instructions of the program counter have an order. For scheduling it is often sufficient to execute the instruction that is most behind (i.e., the first in the ordered list of instructions). After a conditional block (e.g., if-else where both blocks are executed in different contexts), the executions should be joined again. Because the next instructions after the conditional executions are equivalent, the instructions can be joined, and the instruction is shared again. The strategies to maximizes sharing are known as *late splitting* for the late separation of execution paths, and *early joining* for the merging of execution paths [Kästner et al., 2012b] as early as possible. In Listing 4.15, we show a simplified implementation of the variability-aware scheduling mechanism in VarexJ. The shown implementation only searches for the instruction which is most behind by comparing the positions of the instruction in the method saved the methods field `position`.

```java
public void schedule() {
    Conditional<Instruction> pc = initialPC();
    while (!pc.equals(new One(null))) {
        Map<Instruction, FeatureExpr> map = pc.toMap();
        int minPos = Interger.MAX_VALUE;
        Instruction current = null;
        for (Entry<Instruction, FeatureExpr> entry : map.entrySet()) {
            Instruction instruction = entry.getKey();
            if (instruction.position < minPos) {
                minPos = key.position;
                current = instruction;
            }
        }
        FeatureExpr ctx = map.getValue(current);
        Conditional<Instruction> next = current.execute(ctx);
        pc = new Choice(ctx, next, pc).simplify();
    }
}
```

Listing 4.15: Simplified variability-aware scheduling mechanism of VarexJ.

```java
 1  int method(int i) {
 2      int k = 3;
 3      if (FEATURE) {
 4          k = i + k;
 5      } else {
 6          k = k / i;
 7      }
 8      k++;
 9      return k;
10  }
```

Listing 4.16: Example configurable method for variability-aware scheduling.

In Listing 4.16, we show a method with two execution paths which depend on the conditional value of the filed `FEATURE`. For variability-aware execution, Line 4 has to be execution in the context $FEATURE$, and Line 6 in the context $\neg FEATURE$. Furthermore, all other instructions have to be executed for $FEATURE \vee \neg FEATURE$, or simply $TRUE$. We illustrate the execution-traces of the example in Figure 4.3. For simplicity, we only show method statements instead of the corresponding bytecode instructions. We have already illustrated how the execution of bytecode instructions interacts with the method frame in Table 3.1. In the example, all corresponding instructions are executed with the same context, which is not necessarily the case. The traces show that all statements outside of the if-else block are shared between all configurations. Furthermore, it illustrates the order of executions. After the block for FEATURE is executed, the else block is executed directly. After the if-else block has finished, the next instructions are shared again.

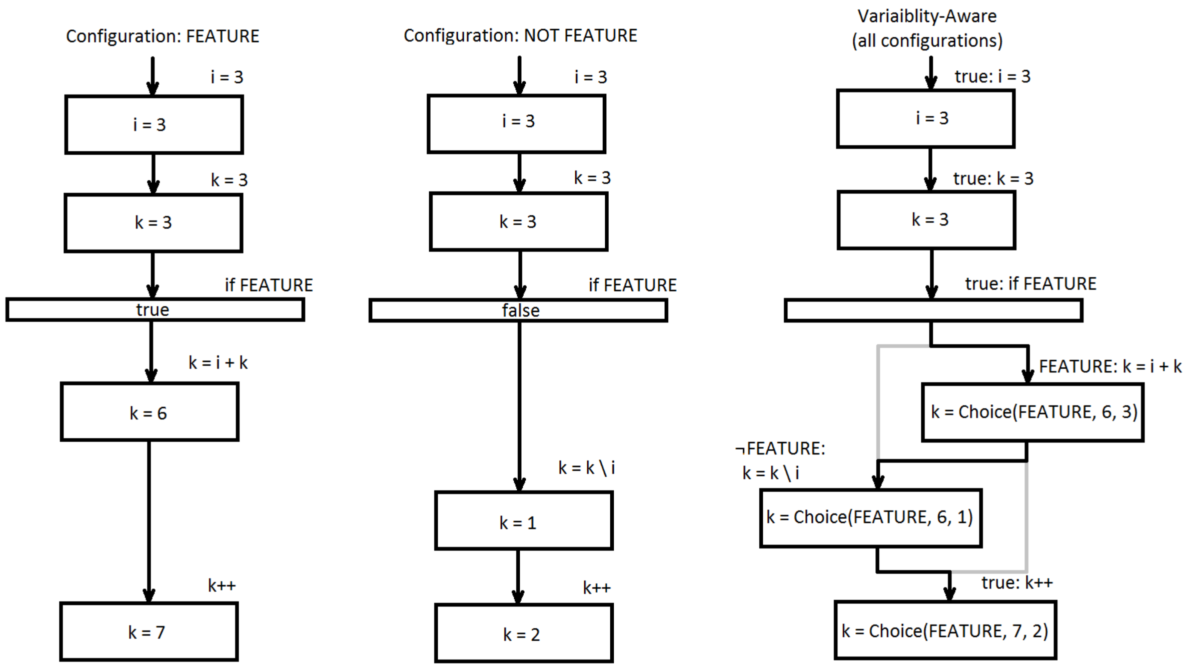

Figure 4.3: Traces of the execution of the example in Listing 4.16 with two configurations. The traces left and middle show the product-based execution with Java Pathfinder. The right trace illustrates variability-aware execution with VarexJ.

Method Return

To simply execute the instruction with the lowest index is usually sufficient. However, there are some instructions which have to be handles differently: return, method invocations and throw instructions. In Listing 4.17, we show a Java source code example which contains multiple return statements. The Listing 4.18, shows the corresponding Java Bytecode. The instruction `IRETURN` pops the top value of the current frame and pushes it to the frame of the invoking method [Lindholm et al., 2014]. Furthermore, the current method frame is discharged, and the frame of the invoking method is the top frame afterward. Because there is only one top frame, the first return instruction cannot be executed before all execution paths of the method are finished. To support multiple returns, return instructions are only executed when there is no other instruction left. Because all return instructions of one method have to be of the same type (e.g., `int`), these instructions (i.e., `IRETURN`) can be merged and executed together at the end of the method's execution.

Exceptions

Similar to a return, a throw instruction (i.e., a thrown exception) pops stack frames. A throw instruction pops the top frame until a catch clause handles the corresponding exceptions [Lindholm et al., 2014]. Because of this exception handling, throw instructions cannot be merged like return instructions, and thus we execute these instructions at the point they appear, and return to the method afterward. After the frames are popped,

```
int method() {
    int k = 1;
    if (FEATURE)
        return k;
    k++;
    return k;
}
```

Listing 4.17: Source code example with multiple return statements.

```
L0      ICONST_1
        ISTORE 1
L1      ALOAD 0
        GETFIELD f
        IFEQ L2
L3      ILOAD 1: k
        IRETURN
L2      IINC 1: k 1
L4      ILOAD 1: k
        IRETURN
```

Listing 4.18: Compiled example with multiple return instructions.

the program counter of the method with the catch clause is set to the first instruction of the catch clause for the context of the thrown exception. Because exceptions can be thrown within a specific context and not all executions of other contexts are finished, we push the popped method frames back and execute the remaining instructions.

Method Invocation

When a new method is invoked a new frame for this method is created, which is then the top frame. To determine which instructions are in a valid context, and to reduce the size of choices, we add the context in which the method is called to the frame. With this method-specific context, the current program counter is often simply a One (i.e., there is only one instruction), because not all methods need to be executed in several paths. Thus, the effort for scheduling and determining the next instruction is reduced.

We also use the method-specific context in the method frames to reduce the complexity of the choices. With the context specific for the method, we do not need to save conditional values that are not valid in this context. Thus, we can reduce the size of conditional values, and the effort for calculations. For example, if a method is called in the context A, it is possible to simplify values, such as Choice(A, 1, 2) to One(1). This optimization has high effect on the performance of the variability-aware execution.

During variability-aware execution, a reference can point to several objects with possibly different types. In the example in Listing 4.19, we show a method which creates a list containing an initial value `init`. The type of `list` is either `ArrayList` or `LinkedList`, depending on the value of `FEATURE`. To execute the method `add` variability-aware, it has to be executed several times, once for each object reference.

```
public List create(int init) {
    List list = FEATURE ? new ArrayList() : new LinkedList();
    list.add(init);
    return list;
}
```

Listing 4.19: Example for method invocation on different objects.

4.3 Specialized Extensions for Variability-Aware Execution

We discussed implementation details on basics for variability-aware execution with reference to Chapter 3. In this chapter, we discuss further specialized details on variability-aware execution. First, we show how the configuration space can be defined to only execute valid program variants. Second, we explain how JPF handles native methods and how VarexJ supports variability-aware execution of these native methods. Third, we present how VarexJ gives aggregated results for an effective analysis of the program's output and of exceptions during runtime.

4.3.1 Specification of the Configuration Space

Variability-aware execution aims to execute all program variants at once. Therefore, our virtual machine needs to know two things about the program and its variability:

- Which values are used as *feature variables*?
- Which combinations of features are *valid*?

To define a value in Java source code as conditional, a new mechanism is required. We decided to mark the boolean values to be handled as conditionals, because we are interested in optional features. Because features would be defined once globally and only have two states we support conditional values for static boolean fields. For definition of fields that should be handled as feature variables, we use the annotation mechanism of Java. In Listing 4.20, we show an example for a feature variable which should be initialized as conditional using the annotation `@Conditional`. A static field is initialized when the corresponding class is loaded [Lindholm et al., 2014]. During initialization, first the initial value is pushed to the operand stack. Then, the top value is used to set the internal representation of the field. As we want to initialize the field with both values (i.e., `true` and `false`), we replace the initial value by a choice. In the example, we pop the original value true and push the choice for both values (i.e., `Choice(feature, true, false)`). For reasoning about features, we create a feature internally that has the same name as the field. Whenever the field `FEATURE` is used it can have both values, true and false depending on the current context of the execution. Thus, our interpreter has knowledge about the variability defined by these feature variables.

```java
@Conditional
public static final boolean FEATURE = true;
```

Listing 4.20: Annotated field to introduce conditional values into the interpreter.

When executing configurable programs, some combinations of features are invalid which can be specified with constraints. When executing a configurable program variability-aware, only valid configurations should be considered. In Listing 4.21, we show a typical example how the configuration space can be reduced to only execute valid configurations. With the method `valid` the selection of features is checked, and only valid selections are executed. This variant of reducing the configuration space is effective; however, the context defined in the method `valid` is always part of the scheduling. Thus, the overhead to decide the satisfiability of the context is high depending of the size of the expression defined in the method `valid`. Because the expression is always part of any choice, also the output contains this expression, which makes it more difficult to aggregate outputs and faults to specific configurations.

```java
@Conditional static final boolean A = true;
@Conditional static final boolean B = true;
static valid() { return A || B; }
public static void main(String[] args) {
    if (!valid()) { return; }
    new Main();
}
```

Listing 4.21: Example for specifying constraints with a method that checks the validity of the current feature selection, to execute only valid configurations.

To specify the valid configuration space, we use the functionality of the library for feature expressions from TypeChef [Kästner et al., 2012a, 2011; Liebig et al., 2013] to set a corresponding feature model. The library allows using dimacs files which represent a feature model in conjunctive normal form. We implemented an automatic export mechanism for dimacs files in FeatureIDE [Thüm et al., 2014b], because the configurable programs we use do not provide such files, and to write it manually is difficult and error prone. FeatureIDE is a framework for feature-oriented software development. One major part of it is a graphical feature model editor to define features and their dependencies. With this editor and an automated export mechanism the creation of dimacs files is simplified.

With the use of dimacs files, the method `valid` is no longer necessary. Because the expression for the configuration space is no longer part of the execution, the expressions in choices are simpler and the choices are smaller, thus the effort for reasoning is reduced. Furthermore, the output only contains the specific context it is executed in, which eases the aggregation of the configuration space for certain defects and outputs.

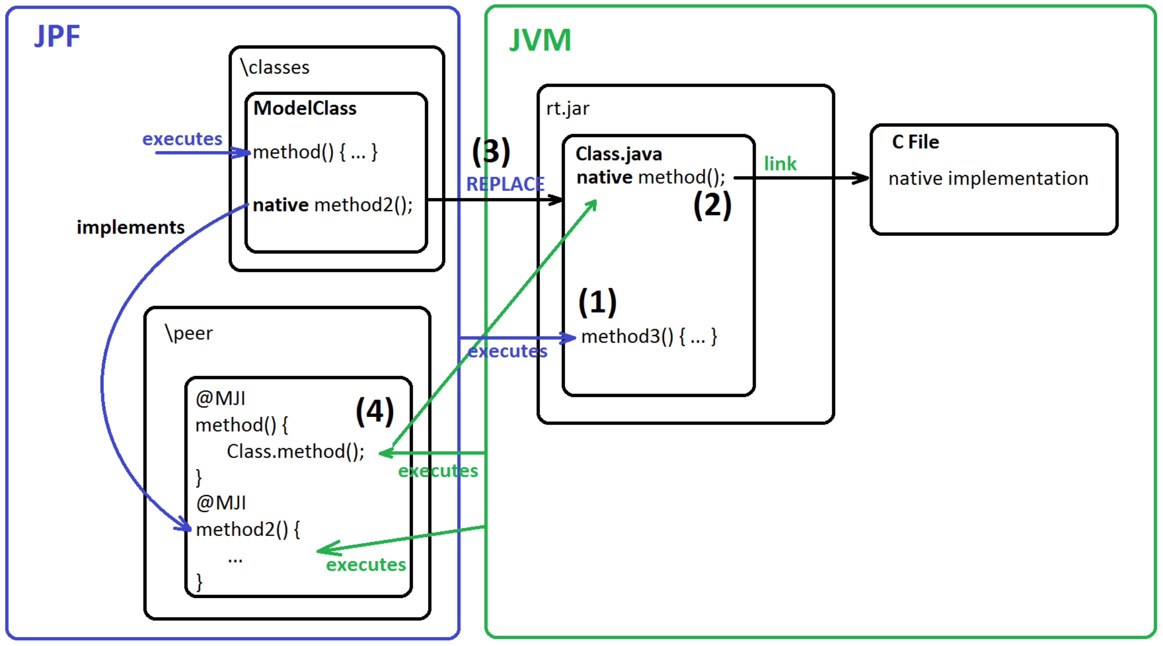

Figure 4.4: Illustration of the support of native method calls in Java Pathfinder.

4.3.2 Peer Methods

Java provides native methods to call platform-dependent methods (e.g., for the file system), usually implemented in C. To be platform independent, each JVM provides own compiled native implementations (e.g., for Windows). Because JPF is executed with another JVM, which already links the native methods to the native implementation, JPF allows reusing these implementations. In Figure 4.4, we illustrate the principle of native method calls in JPF. The library rt.jar of the standard JVM contains all classes for the Java runtime environment, such as java.lang.String or java.io.File. The majority of these classes are Java implementations which can be executed with JPF ((1) in Figure 4.4). However, some implementations are native calls (e.g., for system resources) (2). JPF provides two ways for execution of native methods. Either the complete class is replaces by a model class (3), or the native method call is redirected and executed by the JVM instead of JPF (4). For performance reasons or to hide traces, JPF also has model classes and peer methods (i.e., implementations that replace native calls) which do not require to call the native implementations (e.g., for java.lang.Math). Furthermore, peer methods can directly interact with the environment of JPF.

Sometimes it is necessary to call native methods only within a specific context or with conditional values. Because the corresponding peer method is identified by the method's name and not the method's signature (i.e., the package and original signature is encoded in the name of the peer method), we were able to change the signatures for our purposes of variability-aware execution. We added the context as additional parameter to all peer methods, whether they use it or not. To change all peer methods to support conditional values is complicated, time consuming, and error prone. Thus, we decided to allow peer methods to still support unconditional values. In the case such a method is called with a conditional value, we again throw a runtime exception which indicates that the method has to be lifted. As the peer method might return a conditional value, we also allow conditional return statements. To change the method signature for a peer method, simply the values have to be changed to conditional, because we check the signature before the method is invoked and adjust the parameters automatically if possible.

In Listing 4.22, we show exemplary the peer method `java.lang.Math.max`. This peer method is not interpreted by JPF, but directly executed by the host JVM. Thus, the methods, such as `max` can be called directly. The method `max` does not require to call native implementations; however, this would be possible, as the peer methods are executed by the host JVM. In Listing 4.23, we show the lifted implementation of `max`. The method `max` accepts conditional parameters and returns conditional results. To apply the method to all combinations of `ca` and `cb`, we use a cross-product over both parameters. Furthermore, the method has an additional parameter for the context which might be necessary.

```
@MJI public int max__II__I (MJIEnv env, int clsObjRef, int a, int b) {
    return Math.max(a, b);
}
```

Listing 4.22: Original implementation of the peer method for `java.lang.Math.max` in Java Pathfinder.

```
@MJI public Conditional<Integer> max__II__I(MJIEnv env, int clsObjRef,
        Conditional<Integer> ca, final Conditional<Integer> cb,
        FeatureExpr ctx) {
    return ca.fmap((a) -> {
        return cb.map((b) -> {
            return Math.max(a, b);
        });
    }).simplify();
}
```

Listing 4.23: Lifted peer method for `java.lang.Math.max` in VarexJ with conditional parameters, conditional return type, and a context.

4.3.3 Aggregated Results

Product-based analyses can only produce results that are specific for one configuration. Thus, each configuration has its own outputs. To make general statements about defects and its causes is hardly possible with those product-based results. Especially, to find feature interactions causing defects requires high effort and often only gives in vague results. A benefit of variability-aware execution is that it operates on configuration spaces instead of concrete configurations. Thus, an output or state of the system can always be mapped to specific configuration spaces. In this section, we discuss how VarexJ displays configuration spaces for errors and console outputs.

Aggregation of Errors

A main goal of software testing is to discover defects. With a variability-aware interpreter we have the opportunity to observe all values and their context. Thus, we can effectively test all configurations by providing aggregated results. In Listing 4.24, we show the output of VarexJ for an arithmetic exception caused by division by zero. The method `div` is called with parameters 1 for `i` and 5 for `x`. Both features `a` and `b` subtract 1 of the parameter 1 before `x` is divided by `i`. Thus, a division by zero error is caused when either `a` or `b` is `true`. The output shown in the listing (Lines 21 to 24) is generated by VarexJ. The error report shows the current stack trace at the point the error appeared and the cause of the error ("`division by zero`"). Furthermore, the report displays the corresponding configuration space, highlighted with gray. The propositional formula is shown in conjunctive normal form (i.e., "`&`" has a higher order than "`|`"). The formula is equivalent to $!a \leftrightarrow b$, what exactly covers the configuration space that cause the defect. We see that the third feature `c`, which does not change the context of the error, is not contained in the error report.

```
 1 @Conditional static boolean a = true;
 2 @Conditional static boolean b = true;
 3 @Conditional static boolean c = true;
 4 
 5 @Test
 6 public void testDivByNull() {
 7     try {
 8         div(1, 5);
 9     } catch (Exception e) {
10         e.printStackTrace();
11     }
12 }
13 
14 int div(int i, int x) {
15     if (a) { i--; }
16     if (b) { i--; }
17     if (c) { x += 2; }
18     return x / i;
19 }
20 =======================================================
21  if !a & b | a & !b:
22 java.lang.ArithmeticException: division by zero
23     at cmu.VATest.div(cmu/VATest.java:14)
24     at cmu.VATest.testDivByNull(cmu/VATest.java:6)
25     ...
```

Listing 4.24: Aggregation of errors with VarexJ. The example causes an error if either a or b is true. The output shows the current stack trace, the cause of the error, and the corresponding context.

Aggregated Console Outputs

To test and to detect errors is not the only reason of variability-aware execution. With variability-aware execution it is also possible to check how the values of variables depend on the selections of features and how the features interact. With brute-force execution, a console output contains many duplicate values. With variability-aware execution it is possible to merge these duplicate results to a specific configuration space, and thus provide aggregated outputs for arbitrary variables.

In Listing 4.25, we show a method that modifies a given value `i` depending on the features `a`, `b` and `c`. Finally, the modified value is printed to the console. A brute-force approach would execute all combinations of `a`, `b` and `c`, and thus would display one result for each configuration. With variability-aware execution, we execute the program with conditional values and print this conditional value to the console. With JPF's mechanism of peer methods we were able to modify how `System.out.println` is executed, to print a context specific content. As shown in the output of the listing, `a` and `b` interact with each other. In the listing we show how VarexJ can efficiently merge duplicate results. If `a` and `b` are selected the output is 102.0, if `a` is selected and `b` is not selected the output is 101.0, and if `a` is unselected the output is always 1.0 independent from the selection of `b` and `c`. Because the third feature `c` does not interact with `a` and `b`, the selection of `c` does not change any output. In contrast a brute-force approach would contain many duplicate results (e.g., for `a & b & c` and for `a & b & !c`).

```
@Conditional static boolean a = true;
@Conditional static boolean b = true;
@Conditional static boolean c = true;
void method () {
    double i = 1;
    if (a) {
        i += 100;
        if (b) {
            i++;
        } else if (c) {
            i += 0;
        }
    }
    System.out.println(i);
}
=========================================================
VarexJ Console Output:
    <102.0> : a & b
    <101.0> : a & !b
    <1.0> : !a
```

Listing 4.25: Aggregation of console outputs with VarexJ. The console output shows a conditional value, because the feature selections change the value of `d`. The output shows a mapping of output value to the specific configuration space (highlighted with gray).

A main goal of variability-aware execution and family-based analysis in general is to provide aggregated results. Aggregation eases the detection of feature interactions and increases the value of outputs compared to product-based analysis, and thus simplifies the effort for debugging. We showed how VarexJ provides such aggregated results for errors and console outputs. There are more kinds of outputs which could be lifted to provide aggregated results, such as logging or a conditionally written file content (e.g., with a specialized encoding of the file or with multiple files). However, we leave these tasks for future work.

4.4 Improvements of Variability-Aware Execution

In the previous sections, we presented the current state of VarexJ. In this section, we discuss some open points that we discovered during development which can improve VarexJ and variability-aware execution in general. First, we present limitations of our implementation which reduce the ability of VarexJ to execute arbitrary programs. Second, we discuss possible optimizations that improve the efficiency of variability-aware execution and that give a new insights on sharing.

4.4.1 Current Limitations

A variability-aware virtual machine attempts to execute all configurations, as if they were executed in isolation (i.e., the execution of one configuration is not affected by another configuration). There are cases for which this is not easily possible, because configuration affect each other, especially for execution with side effects, such as writing of files or with interaction with web services. However, also testing of individual systems has these challenges. For example, if a test reads and writes to a specific file. When the test is executed again with another configuration, the file still contains the changes of the other test. With individual execution, it would be possible to reset the file to the original state after each execution. With variability-aware execution this is not easily possible (e.g., the results of one configuration might be replaced by another configuration). New solutions are required to handle such side effects, such as models of the file, separate files for different contexts, or a specialized variability-aware encoding for the file content. Such specialized mechanisms are only necessary if the change of one context effects the execution of another.

Further limitations come with native methods. Because native methods are executed outside of the environment of JPF, the values used for the native execution cannot be conditional. For methods, such `Math.sin`, this is no problem and can be solved with several calls, because the native sinus implementation has no side effects. However, if the method has side effects (e.g., it saves an internal state), multiple executions are no longer possible. To solve the challenges with native calls with side effects, new techniques need to be developed, such as models, workarounds, or a new implementation of the native method.

To apply variability-aware execution to software systems without restrictions, especially caused by side effects, future work has to develop new concepts to overcome the current limitations. Furthermore, variability-aware execution has to be applied to large practically used programs to detect further limitations or scalability problems. Because model checking [Clarke et al., 1999] shares the same challenges, solutions from this research area could be reused.

4.4.2 Optimizations

During the development of VarexJ, we did several optimizations to improve the performance of variability-aware execution, such as a specialized method frame or specification of the global context with a feature model. In this section, we propose some further improvements of sharing and some optimizations to reduce execution times which were out of scope of this work.

Sharing Among Objects

With variability-aware execution, a reference value can point to several objects, as previously discussed and illustrated in Listing 4.19. When a method for this reference is called, the method has to be applied to all references individually. However, when the method implementations for the references are equivalent (e.g., when the objects have the same type), the method could be shared among these objects. In Listing 4.26, we illustrate an example with a reference `list`, which points to two objects of type `ArrayList`. The method `add` needs to be applied to both list. Because both lists are of same type the implementations of `add` for both lists are equivalent. Thus, instructions of `add` could be shared among both lists. This kind of sharing can improve the performance of variability-aware execution, especially when a reference points to many objects, or the called method is expensive. Furthermore, this sharing can be applied for objects of different type that call the same super type method, which is not overwritten.

```
void method() {
    List list;
    if (A) {
        list = new ArrayList(10);
    } else {
        list = new ArrayList(100);
    }
    list.add(1);
}
```

Listing 4.26: Example for possible optimization of variability-aware executions with sharing among objects.

Sharing Among Branches

The same method or instructions might be called from different exclusive branches (e.g., for if-else branches, or switch statements). In this case, it can be useful to join these executions to improve sharing. However, to support this kind of joining, further graph analysis is necessary to know which branches can be joined. In Listing 4.27, we illustrate an example method which calls the same method (`joinMethod`) from different exclusive branches. To share the method can be very efficient; because the method needs to be executed only once and the context of the execution is A∨¬A, what is equivalent to True. This sharing not only reduces the redundant effort, it also reduces the effort for reasoning, because the simpler context True is used.

```
void method(int i) {
    if (A) {
        joinMethod(i)
    } else {
        i++;
        joinMethod(i)
    }
}
```

Listing 4.27: Example for possible optimization of variability-aware executions with sharing among branches.

The improvement with sharing among branches should be higher than the overhead of graph analysis to detect the methods which can be shared. We believe that it can be useful to support this kind of sharing, but the question is how often such exclusive and duplicate method calls actually appear in practice. To answer this question, static code analysis can be used, which checks for pattern in which the same method is called for different branches. However, the results are inaccurate, because such analyses do not know how the methods are executed, and which methods are called multiple times for different contexts. A second way is to log multiple method calls during variability-aware execution. The result is still inaccurate, because the methods are currently not joined and other method calls, which could be joined, might be missed. To get more accurate results the traces of each configuration could be compared to find redundant method calls. To create these traces can be done with variability-aware execution, because they are equivalent to individual execution. Finally, to evaluate the joining mechanism is to implement it and to compare the results for variability-aware execution with joining to the execution without joining. The results should show a difference in the number of executed instructions and required time. In particular, the difference for instructions shows how many instructions can be shared additionally, which indicates the how often method can be shared. The evaluation should also check whether the improved sharing has effect on outputs. For example, if the called method prints to the console, the output only needs to be printed once.

Sharing Among Methods

Calls of the same method can not only happen within the same method, but also within different traces. For example, two objects of different type call the same super type implementation. Especially for metaproduct (a.k.a. product simulators) same methods are called within different traces. A metaproduct is the result of variability encoding, which transforms compile-time variability to runtime variability. The structure and purposes of metaproducts are discussed in detail elsewhere [Post and Sinz, 2008; Apel et al., 2011b; Thüm et al., 2012; Meinicke, 2013]. Several of the programs we used for evaluation are metaproducts for feature-oriented programming, thus we discuss the pitfalls of variability-aware execution of them here. In Listing 4.28, we show an example which calls the same method for different traces. The structure of the example is similar to the branching in metaproducts. The method `joinMethod` is either called directly from the method `splitMethod` or indirectly with the method `specialMethod_FeatureA` if the feature A is selected. In metaproducts, such splitting can happen multiple times. To share the executions of the method `joinMethod` can reduce the effort as for sharing among branches. Again graph analyses are required, to determine whether a method can be shared. As sharing among methods is out of scope of this thesis, we leave the implementation and evaluation to future work.

```
void joinMethod() { ... }
void specialMethod_FeatureA() {
    ...
    joinMethod();
}
void splitMethod() {
    if (A) {
        specialMethod_FeatureA();
    } else {
        joinMethod();
    }
}
```

Listing 4.28: Example for possible optimization of variability-aware execution with sharing among methods.

A variability-aware interpreter is the approach with the highest potential for sharing, compared to other approaches for variability-aware execution. A goal of variability-aware execution is to share computations, and thus to reduce effort and execution times. With sharing among objects, sharing among branches, and sharing among methods, we already presented three approaches that improve the current approaches for sharing. With our new insights on sharing we open a new research field, and terms such as redundancy and sharing get relative. Because sharing can range from one single instruction to sharing of complete methods, it is no longer clear whether an execution is done redundantly. Future work on sharing can lead to new concepts to analyze configurable programs, to reduce redundant calculations (e.g., for code clones), and to define coding conventions that are easier to analyze.

Parallel Execution

To improve the efficiency of variability-aware execution, several parts can be parallelized. Because several executions are done in mutually exclusive contexts, executions do not interact with each other. The map methods apply a function to all elements individually. To support parallel map method, the function can be applied to all elements in an own thread, or several larger parts (e.g., splitting a choice tree into two parts for two threads). With parallelized map functions there is the trade of how large the overhead for creation of tread is compared to a linear execution.

With variability-aware execution also separate method calls can be parallelized (e.g., for different exclusive branched or for method calls on different objects). With parallel execution of whole method, the overhead of thread creation can be neglect, because the effort for interpreting the method is much higher. Thus, parallel execution for separate methods could show a significant speed-up. However, both paralleling strategies require large effort for implementation, because many parts of the JVM might not be able to be parallelized in the current state. Furthermore, parallelization comes with increased memory consumption, which might get a problem, because variability aware-execution already required a lot of memory, compared to execution of one configuration.

Another strategy is to split the configuration space into several parts and execute them in parallel in separate virtual machines. This strategy reduces the effort for every run because only a subset of the whole configuration space is executed. Furthermore, the strategy does not require any changes on the current system. However, the strategy runs several virtual machines in parallel, thus the required memory drastically increases.

Modern CPUs consist of several cores, thus the current single thread variability-aware execution only uses a small part of the actual hardware resources. Thus, we believe that all parallelization mechanisms improve the efficiency. Because execution of different configuration spaces in several threads does not require changes of the current implementation, this strategy can be applied easily. As the sharing potential is reduced, this strategy might not result in high improvements. The other strategies do not reduce sharing, and thus have a higher potential for more efficient execution, but require high effort for realization.

Specialized Peer Methods

To increase the efficiency, JPF already provides peer methods which are execute from the underlying JVM directly. To optimize variability-aware-executions a good strategy is to specialize some methods (e.g., for String or StringBuilder). This strategy was used in Varex with positive results using a specialized data type for string concatenations [Nguyen et al., 2014]. For example, instead of building all combinations for strings with a StringBuilder when new values are appended, the StringBuilder could save the append operations and build required strings when they are needed. An evaluation should check which data types often interact and could profit from such optimization. To specialize peer methods has different effects for different programs, but might be required for scalability [Nguyen et al., 2014].

4.5 Summary

In this chapter, we presented details on our implementation of a variability-aware JVM based on JPF. We discussed our decision for JPF as best suiting JVM for our purposes. Then we explained our general procedure for lifting of JPF. We discussed details and design decisions of parts of our implementation, namely method frame, bytecode instructions, scheduling, specification of the configuration space, and peer methods. Furthermore, we presented our solutions to aggregate results and errors with VarexJ. Finally, we stated possible improvements of variability-aware execution which were out of scope of this work. In particular, we discovered several new approaches to improve sharing, which give to new insights on sharing and redundancies in analyses of configurable programs.

5. Evaluation

In this section, we evaluate the scalability and the efficiency of variability-aware execution with VarexJ. Furthermore, we present some new ways to analyze interactions in configurable systems that are possible with variability-aware execution. With our evaluation we want to answer these two questions:

- **(RQ1)** Efficiency: Is variability-aware execution more efficient than product-based approach? How much time does it take to execute configurable programs, and how efficient can executions be shared among configurations?
- **(RQ2)** Sharing and Interactions: How large are interactions among features and how many different values does one variable take?

To answer these research questions, we first introduce into the configurable systems we use for our evaluation in Section 5.1. In Section 5.2, we discuss our framework we use for our measurements. We evaluate the efficiency and effectiveness of our implementation VarexJ in Section 5.3 by comparing the time for executions and number of instructions for variability-aware execution to product-based execution. In Section 5.4, we evaluate the interactions appearing in our target systems to present the potential of a variability-aware interpreter to analyze feature interactions. In Section 5.5, we discuss the results of our evaluations and discuss threats to validity.

5.1 Target Systems

For the evaluation of VarexJ, we use 10 configurable systems. Nine of these systems were previously used for the evaluation of other variability-aware approaches based on JPF [Apel et al., 2013d; Kim et al., 2012]. Thus, the systems are implemented with Java and are known to be directly executable with JPF (i.e., all native methods are available). We used these systems for the development of VarexJ to incrementally lift JPF, to handle conditional values. Furthermore, we are able to compare the efficiency of our implementation with the previous variability-aware approaches.

The system *BankAccount* is a simple bank management system. Thüm et al. [2014c] used the system for family-based model checking including runtime assertion checking with JPF-BDD [Apel et al., 2013d] and family-based theorem proving with KEY [Beckert et al., 2007]. The system is developed for design of contract [Meyer, 1992]; however, we ignore the contracts for our evaluation and do not compile runtime assertions for our executions. We used the system in the initial phases of the development of VarexJ, because of its simplicity.

We use several programs from the evaluation of JPF-BDD [Apel et al., 2013d]. We use the product lines *AJStats*, and *ZipMe* available at the FeatureHouse repository [Apel et al., 2013b].[1] Furthermore, we use the systems *E-Mail* [Hall, 2005], *Elevator* [Plath and Ryan, 2001], and *Mine Pump* [Kramer et al., 1983]. In the evaluation with JPF-BDD, Apel et al. [2013d] compared the effectiveness to detect defects with τ-wise sampling strategies to the variability-aware approach.

All product lines from the evaluation of JPF-BDD and the product line BankAccount are implemented with feature-oriented programming. Feature-oriented programming is an extension of object-oriented programming, where each feature is implemented in separate modules [Prehofer, 1997]. With a configuration, these modules are generated to a program variant which only contains the modules of the selected features. Because feature-oriented programming implemented variability at compile time, a metaproduct is used to simulate runtime variability [Apel et al., 2011b]. We directly reuse the metaproducts from the previous evaluations generated with FeatureHouse [Apel et al., 2013b]. Furthermore, we directly use the metaproducts for product-based executions.

As discussed in Section 4.4.2, metaproducts have properties which are not optimal for variability-aware execution. Thus, we evaluate further systems which implement runtime variability. From the evaluation of shared execution [Kim et al., 2012], we reuse the product-line for graph analysis *GPL* [Lopez-Herrejon and Batory, 2001]. GPL is a widely used configurable system for several kinds of analyses. The system implements several algorithms for graph analysis, such as for cycle detection or shortest path calculations. Furthermore, we use the programs XStream [Walnes, 2014] and JTopas [JTopas, 2014], for which Kim et al. [2012] introduced runtime variability for the evaluation of shared execution.

We also evaluate a software system which is not developed as product line, because it might show different results. The system *QuEval* is a framework for quantitative comparison and evaluation of high-dimensional index structures [Schäler et al., 2013]. For comparison, QuEval provides several implementations (e.g., for index structures) which can be activated and deactivated with runtime parameters.

In Table 5.1, we summarize the programs we use for our evaluation. We calculated the lines of code and number of bytecode instructions with EclEmma [Hoffmann, 2009], an Eclipse plug-in for code coverage measurements. The lines of code do not contain

[1] http://fosd.net/fh

System	LOC	Instructions	Features	Products	Input
AJstats	8854	35235	20	32768	Variant 2
BankAccount	965	4217	10	144	-
E-Mail	644	2075	9	40	-
Elevator	730	4816	6	20	Spec 1, Var -1
GPL	715	2971	22	146	Random 4
JTopas	715	14528	5	32	Many Comments 2
Mine Pump	296	1687	7	64	-
QuEval	3109	18294	30	1135	-
XStream	9224	38234	7	128	0 Com, 30 Var
ZipMe	2762	12901	8	10	Variant 2

Table 5.1: Overview on configurable systems we use for evaluation of VarexJ. Lines of code (LOC) do not contain empty lines or comments. For the projects JTopas and XStream we did not count the lines for the tests, because there is more code for tests than for actual source code.

empty lines or comments, thus the numbers differ from previous evaluations for JPF-BDD [Kästner et al., 2012b] and shared execution [Kim et al., 2012]. For our purposes, the number of instructions is a more accurate value for the size of the program and is more interesting. The systems only range up to 30 features, but even small numbers of features can cause a high number of program variant, as for AJStats which has 32,768 distinct configurations. The system GPL has seven, and QuEval has six abstract features, which are used to model variability, but do not affect the system at implementations level [Thüm et al., 2011]. In the systems we use, the abstract features do not increase the number of configurations. However, they are used to reason about valid configurations and thus increase the effort for variability-aware execution with VarexJ.

To compare our measurements with previous results for variability-aware executions [Kim et al., 2012; Apel et al., 2013d], we show the input values used for the program executions. For the programs BankAccount there are no further parameters to modify the program. Because the system QuEval is never used before for variability-aware executions, we have a fixed main method. The systems E-Mail, Elevator, and Mine Pump use model checking to simulate random executions (i.e., random actions are performed). Because, we do not use these model checking abilities we use a fixed order of actions for these three systems. Thus, our results for these systems are only indirectly comparable with the results of the evaluation of JPF-BDD [Apel et al., 2013d].

The target systems we use are all small and only contain a minimal amount of configuration options. The main reason that we use these systems is that all other systems we tried require additional peer methods for execution with JPF and require further effort for lifting of VarexJ. The systems we use required none or only some effort for peer methods (i.e., for lifting of existing and introduction of new peer methods).

5.2 Evaluation Framework

To answer our research questions, we designed an evaluation framework that:

- creates configurations for the product-based executions,
- executes the programs with a standard JVM, JPF, and VarexJ,
- measures comparable times,
- counts executed instructions,
- measures memory consumption, and
- evaluates sharing and interactions of the variability-aware executions.

Configuration of the Target System

To define a configuration, each system contains a class which defines *all* configuration options and the validity of the selection. In Listing 5.1, we illustrate am example class for the configuration. It contains two annotated global options, `option1` and `option2`. Furthermore it contains a method `valid` that returns weather the selection is valid. In the example, at least one of both options has to be selected. To create the expression for the validity, we use the export mechanism of FeatureIDE [Thüm et al., 2014b] that can saves a feature model in cnf format with Java modifiers.

```
public class Configuration {
    @Conditional
    public static boolean option1 = true;
    @Conditional
    public static boolean option2 = true;
    public static boolean valid() {
        return option1 || option2;
    }
}
```

Listing 5.1: Example file representing configurations for a target system.

To execute one specific configuration, we parse the arguments of the main method of the system to select the configuration options. After the configuration is specified, we execute the application. In Listing 5.2, we illustrate an example class for the selection of configurations. First the arguments are parsed, then the selection is checked, and if the selection is valid the application is executed.

```java
public class Main {
    public static void main(String[] args) {
        if (args.length > 0) {// do not select features for VarexJ
            Configuraiton.option1 = Boolean.valueOf(args[0]);
            Configuraiton.option2 = Boolean.valueOf(args[1]);
        }
        if (!Configuration.valid()) {
            throw new RuntimeException();
        }
        new Main();
    }
    public Main() {
        // run application
    }
}
```

Listing 5.2: Example file for selection of a configuration and execution of the application.

We use reflection to create all configurations with our framework. At first, we parse the configuration class to find annotated fields. To create all configurations, we use a brute force algorithm that tries all possible combinations of selections. With the `valid` method the invalid configurations are ignored. Because the systems QuEval and AJstats have over 1,000 configurations we only execute a random subset of configurations, and extrapolate the estimated time for all configurations based on the average execution time. To create a random configuration we try out random selection in a brute-force fashion. The brute-force algorithms are sufficient for our systems, because of their small amount of features. Finally, the selection of features is used as program parameters for the product-based executions. For variability-aware executions a selection of features is unnecessary.

Execution

For product-based and variability-aware executions, we call the programs via command line. We create a new virtual machine for the execution with the host JVM. To execute the system with JPF or VarexJ, we also create a JVM that starts JPF respectively VarexJ. For the product-based executions this procedure is applied to all configurations. To evaluate our designs for choices (i.e., map choice and tree choice), we execute VarexJ for both implementations. We always execute the program with a dimacs file (except of AJstats which only contains optional or mandatory features), because using a dimacs file to specify a feature model is better by orders of magnitude. To reason about valid configuration we support both, SAT solvers and BDDs. However, because the implementation with SAT solvers does not scale we always use BDDs in our evaluation.

Measure Comparable Times

To compare the effort for execution we could measure the time from the first configuration until the last configuration is executed. This time would represent the real time it takes for the evaluation of one approach, but this time would also contain the time to find configurations. Furthermore, it contains the high overhead of creating the virtual machines. For some applications it might be required to create a separate virtual machine to reset the systems [Bell and Kaiser, 2014], but because the effort is too high compared to the actual execution of the program we exclude these times. The statistics calculated by JPF contain a separate time for the execution that does not contain the effort of creating the virtual machine. However, the measured time contains a lot of overhead for class loading and initializing of the application. This initial effort has negative effect to the time for product-based executions with JPF compared to VarexJ. As we want to measure fair comparable times for all three virtual machines, we measure the actual time for the execution of the program. Therefore, we add the time measurement to each application and print the spend time to the console which can be read by our framework. In Listing 5.3, we illustrate the measurement for comparable execution times that only measure the time spend for execution of the application.

```java
public class Main {
    public static void main(String[] args) {
        // select configuration options
        long start = System.nanoTime();
        new Main();
        long end = System.nanoTime();
        System.out.println("Time:" + (end - start));
    }
    public Main() {
        // run application
    }
}
```

Listing 5.3: Example file for measuring of execution times. The time is printed to the console which can be read by our framework.

The execution times differ for each execution of the same program. All executions are done 10 times to avoid this computation bias. As final result, we use the average time of these 10 executions. To measure the actual performance for the host JVM would require further effort. The time for execution with the host JVM can be better than what we measure when we create a new virtual machine for each run, because of just-in-time compilation, optimization in the virtual machine, and garbage collection. As JPF does not contain optimizations such as just-in-time compilation and the overhead of JPF itself as virtual machine is high, the time we measure for JVM are sufficient for our purposes.

Further Measurements

To evaluate *sharing of instructions* among configurations, we count the number of executed instructions for product-based execution with JPF and VarexJ. The statistics of JPF already contain a counter for instructions. This counter also contains the instructions for initialization. Because the initialization phase does not contain any variability, we reset the counter at the point the application is actually started. With this counting, the amount of executed instructions matches also the time we measured. For execution with the JVM we do not count the executed instructions, but they should match the number executed by JPF.

Variability-aware execution increases the *memory consumption* for execution compared to execution of only one variant. Again JPF contains a measurement for the maximum required memory which we can reuse for the execution with JPF and VarexJ. We do not measure the required memory for the JVM, but it should be much smaller than the memory required for JPF.

In Listing 5.4, we show an example output of the statistics calculated by JPF. It contains several statistics, such as required time and loaded code. Our framework parses the statistics printed to the console by JPF and reads the number of instructions and the maximal required memory. As discussed, the time calculated by JPF is not representative for our purposes.

```
==================================================== statistics
elapsed time:       00:00:02
states:             new=1,visited=0,backtracked=0,end=1
search:             maxDepth=0,constraints=0
choice generators:  thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,
                    reschedule=0), data=0
heap:               new=426,released=43,maxLive=0,gcCycles=1
instructions:       3075
max memory:         173MB
loaded code:        classes=65,methods=1295
```

Listing 5.4: Example statistics calculated by Java Pathfinder. The statistics we reuse are highlighted with gray.

To answer RQ2, we want to measure how instructions are shared and how interactions appear. We count the number of features for the context in each instruction to evaluate how instructions are shared. This means when the number is zero, the context is simply True, and thus the instruction is highly shared. When the context contains one feature the instruction is shared among all configurations where exactly one feature is selected or deselected, and the other feature selections can be arbitrary or are implied. However, the context might contain features which are implied by other feature selection in the context. Thus, the number we calculate can be higher as the actual number of required feature selections.

We analyze the choices that are created to evaluate how values are shared among configurations. Again we count the number of features required to define a conditional value (i.e., we count each feature only once). Furthermore, we calculate the size of the conditional value. That means we count each leaf for a tree choice and each entry for a map choice. The higher these values are the higher is also the effort for variability-aware execution. They also indicate higher order interactions which are likely to be missed with product-based strategies, such as τ-wise interaction testing.

5.3 Efficiency of Variability-Aware Execution

The main goal of variability-aware execution is to provide efficient testing of configurable systems. To evaluate efficiency, we compare the time for variability-aware execution to product-based execution with JPF, and directly with the host JVM. For evaluation we use the JVM from Oracle version 7 update 55. We performed all experiments on a Windows 7 laptop with an Intel i7-620M CPU with 2.67GHz, 4 cores, and 6GB RAM.

With our evaluation we compare product-based testing of all products with variability-aware testing. We evaluate the time for τ-wise sampling for $\tau \in \{1, 2, 3\}$. Therefore, we did not create explicit configurations, but calculated estimated values based on the average time for execution of one product. To create the number of required products to fulfill τ-wise coverage, we used the algorithm ICPL that can calculate configurations for 1, 2, and 3-wise coverage based on feature models [Johansen et al., 2012]. Furthermore, the algorithm guarantees 100% coverage. The number of configurations by ICPL might not be the absolute minimum for a τ-wise coverage. However, the difference is minimal, especially because of the small feature models we use. To avoid computation bias, we executed each experiment 10 times and use the average execution times as final result. Furthermore, we executed VarexJ for both implementations of conditional values, map choice and tree choice.

In Figure 5.1, we show the results of our evaluation. The diagram shows execution times of product-based executions and variability-aware executions relative to the execution of all products with JPF (i.e., brute-force executions with JPF is always 100%, and 5% means that the execution is 20 times faster than JPF). The relative times are shown on a logarithmic scale, as our measurements differ by orders of magnitude. The time for τ-wise execution is an estimated value based on the average execution time of one product. For τ-wise, we do not require exact measurements, because the number and the set of configurations differs among sampling algorithms. The times time for execution of all products for QuEval and AJstats is extrapolated based on the average of 100 random configurations, as the execution of all configurations is too time consuming and does not give more meaningful results.

The implementation of conditional values with map choice contains each value only once and is a more memory space saving implementations than tree choice. Because map choices do not contain redundant values, also the amount of applied calculations with map functions (e.g., mapf) is reduced compared to tree choices. It seems that the effort to create map choices and the increased effort for reasoning causes map choice to be less efficient than tree choice by a factor between 1.3 and 2.5, except of for XStream where map choices are minimal better than tree choices. For comparisons with product-based executions, we refer to the best times measured for variability-aware execution, as we evaluated VarexJ with two implementations for conditional values.

Comparison with Java Pathfinder

Except for the results for ZipMe, Varex is always faster than brute-force execution with JPF. Because ZipMe only consists of 10 products, the advantage of variability-aware execution to share calculation has a too small effect compared to the overhead

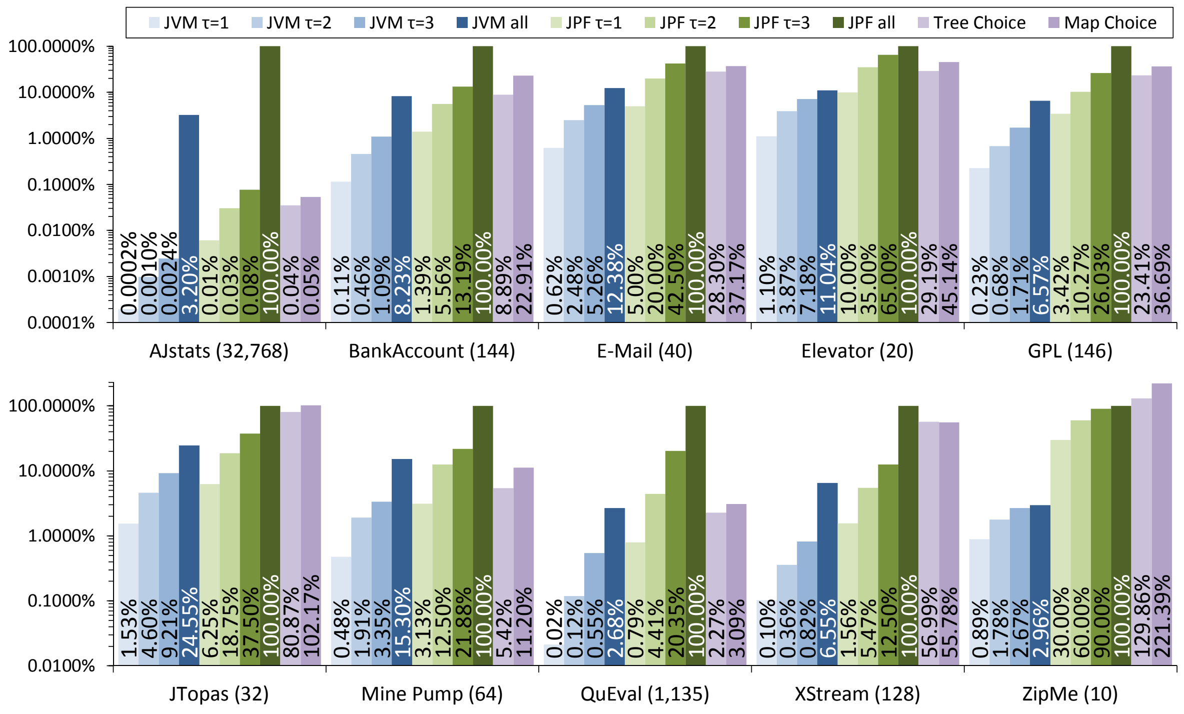

Figure 5.1: Execution times for product-based testing with the JVM and Java Pathfinder, and variability-aware execution with VarexJ using tree choices and map choices as conditional values. The times are shown on a logarithmic scale. All execution times are relative to the product-based run with Java Pathfinder.

of calculation with conditional values. Furthermore, except for JTopas, XStream, and ZipMe, VarexJ is also faster than 3-wise sampling. For AJStats, Elevator, Mine Pump, and QuEval it is even more efficient than 2-wise sampling. The execution time for 1-wise sampling is always faster than VarexJ. The more products are needed to be executed the better is the efficiency of variability-aware execution. For the system QuEval VarexJ takes only 2.27% compared to JPF, and for AJStats it takes just 0.04% of the execution with JPF.

Relative execution times contain the exponential number of configurations and to execute all configurations is unusual and impractical. To compare variability-aware execution with arbitrary product-based execution, we show the overhead compared to average time for execution of one product in Figure 5.2. The diagram shows that the overhead for variability-aware execution rather depends on the structure and implementation of variability in the systems, than the number of features and configurations. For the system AJstats with 32,768 configurations VarexJ only needs the same time as the execution of 11.52 configurations. On the other hand, for XStream with only 128 configurations, VarexJ already requires the same time as execution of 71.4 configurations. Future work needs to investigate which kind of variability causes overhead to improve variability-aware analyses and to provide rules for implementation of variability that can be analyzed efficiently.

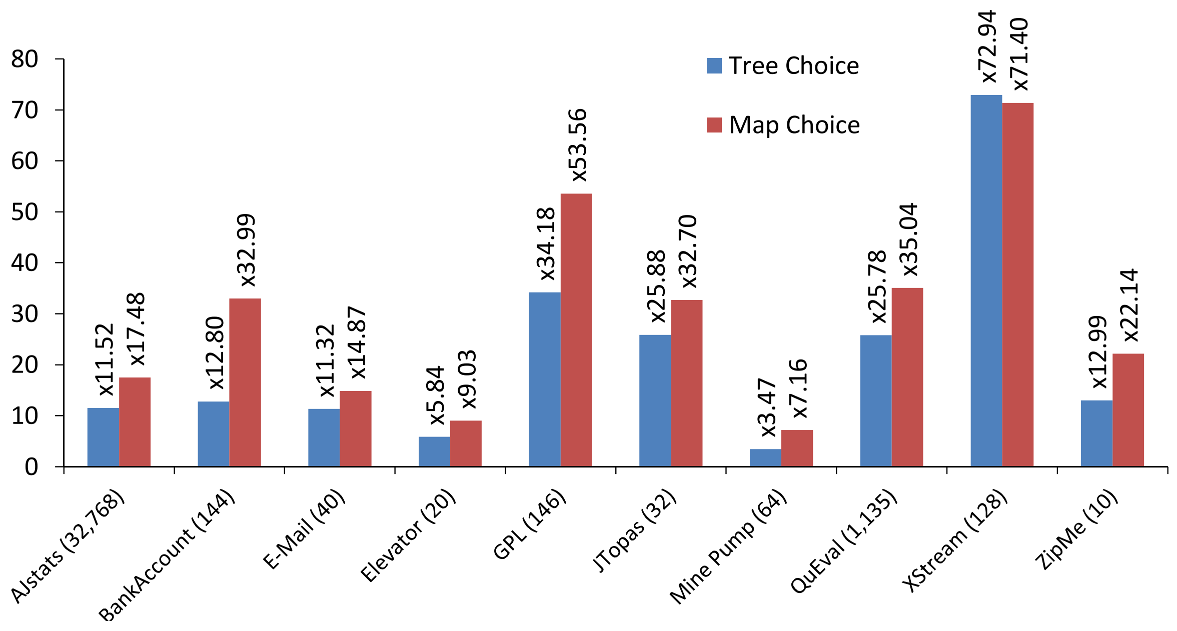

Figure 5.2: Overhead of variability-aware execution compared to the average time for execution of one product with Java Pathfinder (i.e., one product equals x1).

Comparison with the Host JVM

The measurements of the execution time with the host JVM in Figure 5.1 need to be interpreted with caution, because the executed programs and test cases are all relatively small. Furthermore, optimization mechanisms, such as just-in-time compilation cannot be applied and rather have a negative effect to the execution time. Thus, the relative times of execution with the host JVM can be much smaller. However, because of the overhead of JPF, we did not expect that VarexJ could outperform the host JVM. The execution of all products with the JVM is faster than variability-aware execution for the systems BankAccount, E-Mail, Elevator, GPL, JTopas, XStream and ZipMe by up to 33.5 times. Furthermore, τ-wise sampling is always faster than VarexJ (for $\tau <= 3$). For the system Mine Pump with only 64 configurations VarexJ is already faster by a factor of 3. This efficiency seems to be caused because the test case is relatively small. For the larger systems VarexJ again outperforms brute-force testing. For the system QuEval VarexJ is faster by a factor of 1.17, and for AJstats by a factor of 87.4.

Sharing of Instructions

A main indicator how efficient VarexJ can share executions is the amount of required instructions. We show the relative amount of required instructions for product-based execution with JPF and variability-aware execution with VarexJ in Figure 5.3. Because the implementations map choice and tree choice do not affect the implemented scheduling mechanism, we have only one value for VarexJ. The diagram shows that VarexJ can share instructions and reduce the number by at least 80%, even for ZipMe with just 10 configurations. The more configurations a system has, the more instructions can

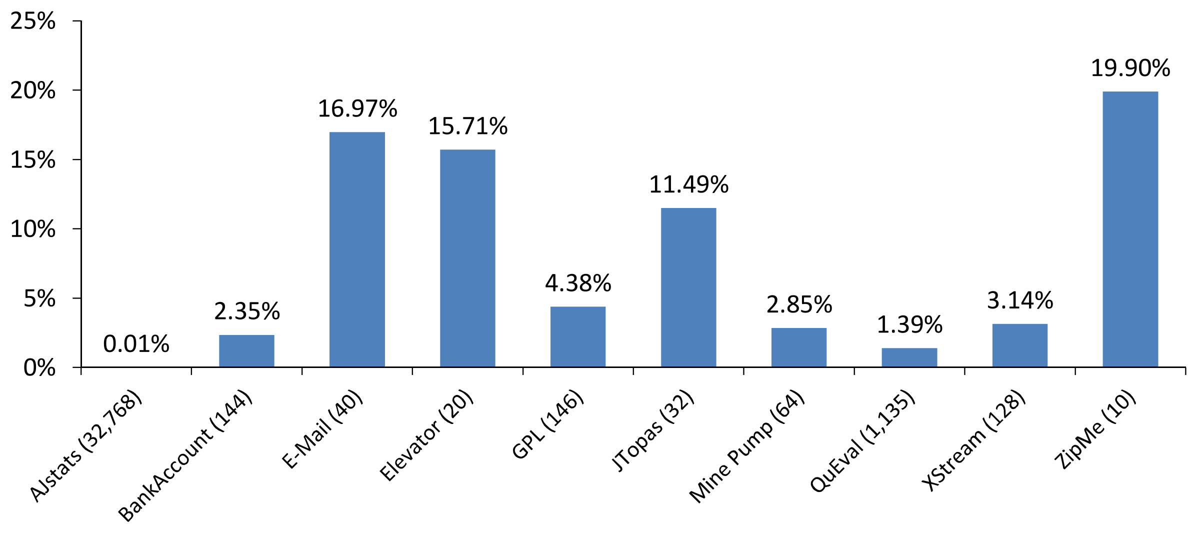

Figure 5.3: Instructions for variability-aware execution with VarexJ relative to instructions for product-based testing with Java Pathfinder.

be shared. For BankAccount the number of instructions is reduced by already 97.7% for only 144 configurations. For the larger systems the sharing is even better. The instructions for QuEval can be reduced by 98.61% and for AJstats by 99.99%.

Another measurement for sharing of instruction is the relative overhead of variability-aware execution compared to the average instructions required for one configuration. In Figure 5.4, we show how many configurations can be executed with the number of instructions executed with VarexJ. The systems Mine Pump and ZipMe require an overhead of less than one additional configuration. The systems AJstats, BankAccount, Elevator, JTopas, and XStream require instructions for between 3 and 4 configurations, and the systems E-Mail and GPL between 6 and 7. The most overhead is required for QuEval with instructions for 15.78 configurations. The numbers show how big the overhead is to execute the same code as executed by all products. Because the system QuEval has a lot of alternative implementations, the overhead is highest compared to the overhead of the other systems.

Reducing the instructions to execute also reduces to the required execution time. However, the actual efficiency is much lower than sharing of instructions, because variability-aware execution needs to calculate with conditional values and needs to reason about configuration spaces. At the point where the overhead of conditional values is compensated by the reduction of required instruction, variability-aware execution is more efficient than product-based execution. Due to of the configuration space explosion variability-aware execution gets more efficient very fast, because executions can be shared more efficiently within huge configuration spaces.

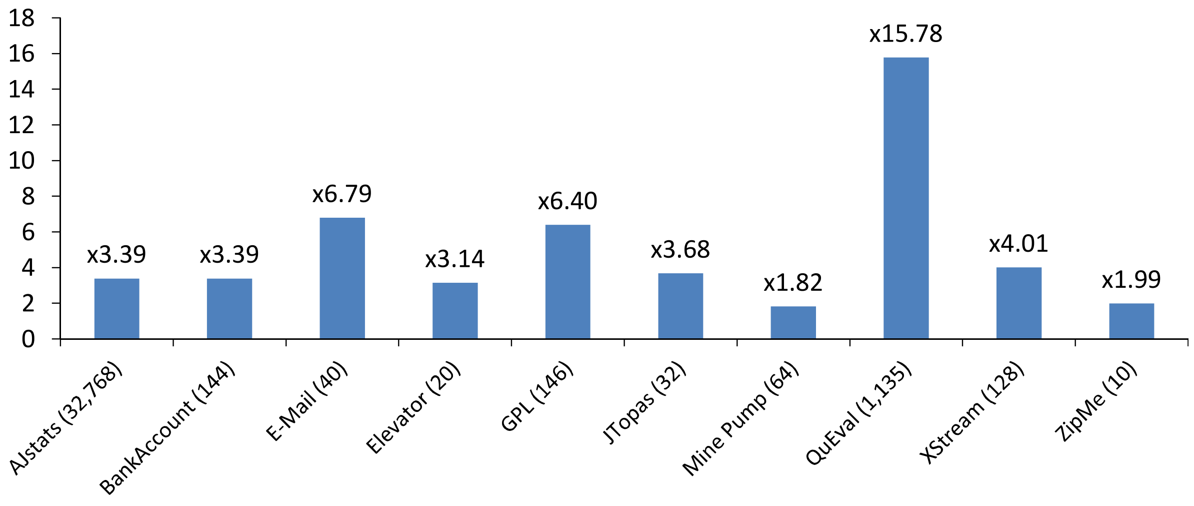

Figure 5.4: Overhead of executed instructions for variability-aware execution compared to the average instructions for execution of one product (i.e., one product equals x1).

Memory Consumption

Variability-aware execution saves all intermediate values off all configurations at the same time. Thus, the memory consumption can increase drastically compared to a execution of only one configurations. Because data can be shared among configurations, the amount of unique values does not rise as much as the number of configurations.

In Figure 5.5, we show our measurements of maximal required memory for the product-based execution with JPF and the execution with VarexJ. As to expect, the required memory for variability-aware execution is always higher than for product-based executions. For execution with JPF the required memory is almost similar for all systems at 90 MB, except of for XStream with 465 MB. For the simple systems BankAccount, E-Mail, Elevator, and Mine Pump, the required Memory is not much higher as for product-based execution, because these four systems do a lot of calculations, but do not create many new objects. For the other systems, the required memory is already in the gigabyte range with up to 2,294 MB for XStream. The positive observation is that the required memory does not always increase with the number of configurations, especially because values can be shared among configurations. The measurement of ZipMe shows that the required memory can be higher that the number of configurations. We think that there are some problems with the current implementation of VarexJ specific to garbage collection and that data is not released correctly when it is no longer required. Kim et al. [2012] already discussed this problem for shared execution and proposed some specialized solutions. In particular, the experiments with QuEval did not scale for larger data sets. To develop specialized variability-aware garbage collection is out of scope and should be a main part of future work, because the memory consumption seems not to scale for systems that require more resources.

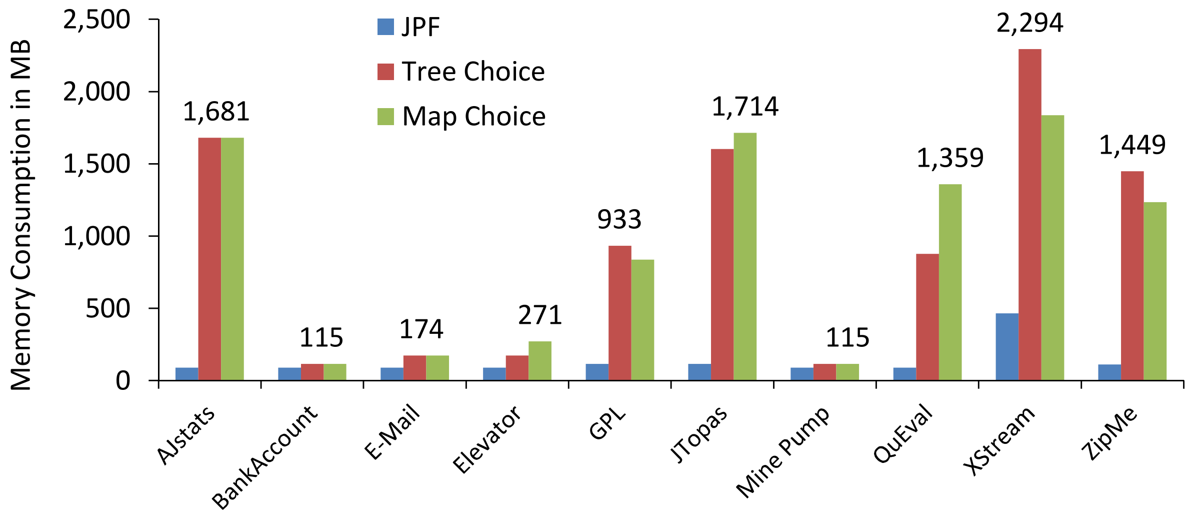

Figure 5.5: Measurements of maximal required memory for product-based execution with Java Pathfinder and variability-aware execution with VarexJ. The measured memory for VarexJ highly differs between each run. Thus we show the highest value of 10 runs.

5.4 Sharing and Interactions

To evaluate how instructions and data are shared we analyze the conditional values used by VarexJ. In contrast to the previous evaluation, where we evaluated the efficiency of VarexJ, we analyze the properties of the systems.

Sharing and Interactions in Computations

We measured the context in which each instruction is executed, to evaluate how good instructions can be shared among configurations. Therefore, we counted the number of features involved in the context (i.e., f(True) = 0, f(A) = 1, and f($\neg A \vee (A \wedge B)$) = 2). Smaller numbers of features in a context of an instruction indicate that more configurations can share this instruction. If the context is True, then this instruction is shared among all configurations. If the context is more specific, the instruction is shared for all configurations where the context holds.

In Figure 5.6, we show the distributions of the context sizes for each system. A similar evaluation was done for the plug-in application WorldPress with Varex [Nguyen et al., 2014]. For WorldPress the majority of instructions were executed for True or within contexts that contain only one feature. As WorldPress uses plug-ins which usually not interact with each other as much as runtime variability, this is not surprising. We measured similar results only for AJstats, GPL, and ZipMe. In particular, GPL uses runtime variability to activate single analyses (e.g., counting of nodes). After such a analysis is done, the next analysis is executed. Thus, the context of the execution of each analysis is only the corresponding feature. For WorldPress a lot of instruction were shared among all configurations. We could not reproduce these observations (only some

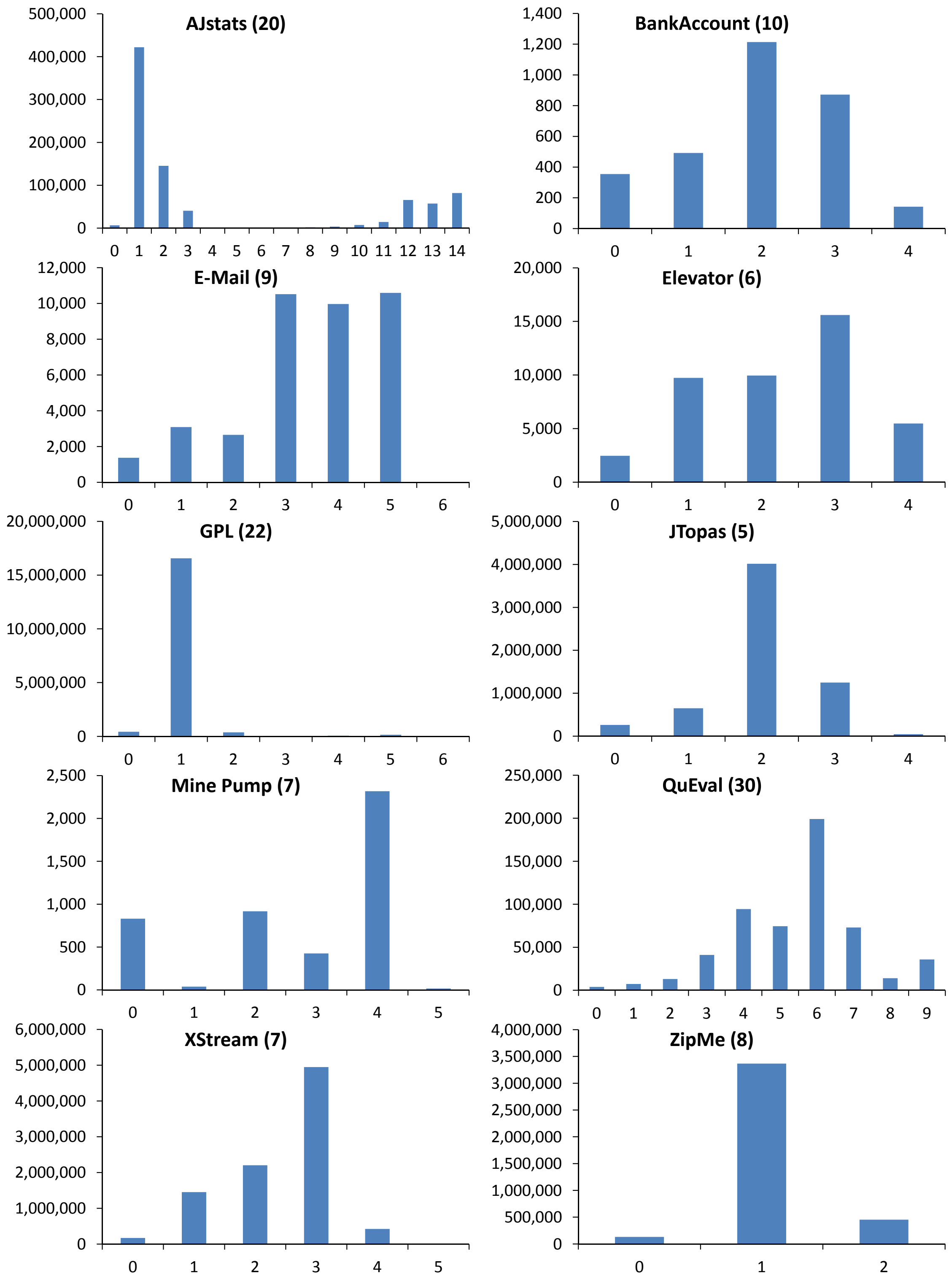

Figure 5.6: Distributions of context sizes of executed instructions for each system. The number of features of the system is shown in the braces after the names.

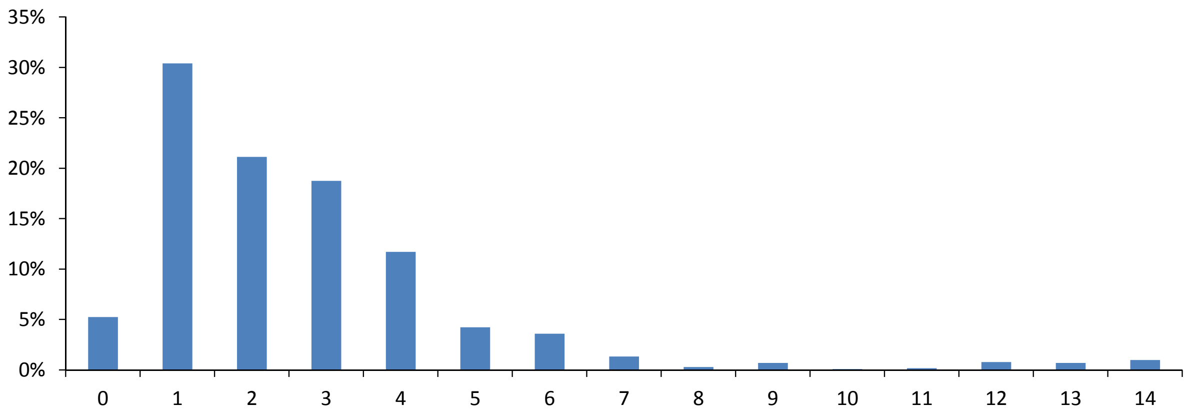

Figure 5.7: Distribution of the context size of executed instructions over all systems. The individual distributions are relative for each system (i.e., each system covers exactly 10% of the overall distribution).

instructions have the context True, e.g., for initialization of the program), because our systems generally require at least one feature to execute a specific functionality. For the other systems the number of features in the context is usually higher. This higher number indicates that several features influence the executions of each other. The diagram of QuEval shows that most of the instructions were executed within a context with six features.

In Figure 5.7, we show the distribution of the context sizes over all systems. To compare the sharing among all systems, we use relative distribution for each system). The chart shows that most instructions are executed where exactly one feature is specified to be selected or unselected. Furthermore, over 50% of instructions are specified within a context of less or equal to two features. This means that most instructions can be shared over huge configuration spaces. Yet, the distribution must be interpreted with caution, because the systems AJstats, GPL, and ZipMe take together around the half of the instructions executed within a context with one feature. Generally, the number of feature in a context is low and thus most of the instructions can be shared for huge configuration spaces.

Most instructions can be shared among many configurations, but the effort for one instruction highly differs for variability-aware executions. Because the instructions need to calculate with conditional values, the effort for one single instruction can be immense depending on the size of the values, and the effort for reasoning. To reduce these effort, we implemented optimizations, such as a specialized method frame. Future work should investigate which data structures, are better for calculations with conditional values.

Sharing and Interactions in Values

To evaluate interactions among features, we counted the number of features required to define the context of conditional values. As for the evaluation of instructions, we count each feature only once. We only measured the highest degree of interactions in

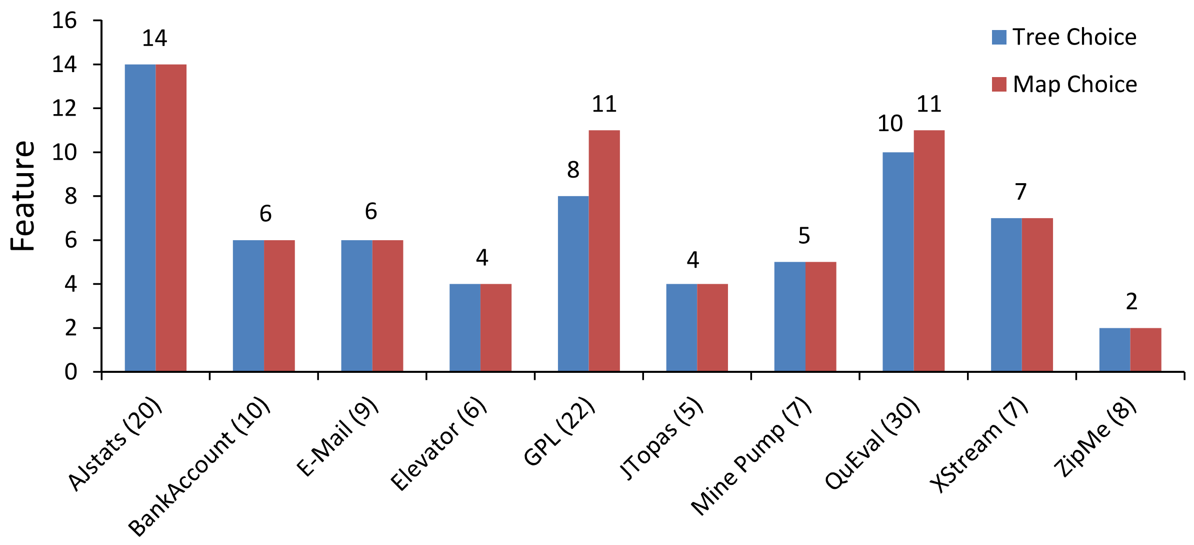

Figure 5.8: Maximal number of features required to define conditional values. The number of features of the system is shown in the braces after the name.

each system by observing conditional values, because values are scattered in the virtual machine in several places, such as the frame, the heap, and fields, what makes it more difficult to observe individual values.

In Figure 5.8, we show the size of the contexts for each system for tree choices and map choices. Except of ZipMe, all systems have relatively high maximal interactions. The maximal interaction in the system XStream contains already all seven features. The highest interaction is contained in the system AJstats with 14 features. Because choice trees can split the context with their structure, they can require less features than choice maps as shown for GPL and QuEval. The degrees of interactions show that some interactions might be missed by τ-wise sampling, as the generation and analysis of the corresponding configurations gets unpractical for $\tau > 5$ [Petke et al., 2013].

The number of involved feature is only one sight on the interaction of values, and an interaction of n features does not require 2^n unique values. Thus, we measured the maximal number of values in conditional values. For map choice, we counted each value. For tree choice, we counted each leaf. The number of unique values for tree choice can be smaller than the measured value and should match the value measured for map choices.

In Figure 5.9, we show the maximal sizes of conditional values for each system. As discussed, the number of features in values is not directly connected to the number of values. For the systems, Mine Pump and ZipMe the number of unique values is with maximal three relatively small. When comparing our results for Mine Pump with the evaluation of [Apel et al., 2013d], we need to remind that we did not use model checking to simulate random actions. Thus, the degree of interactions we measured is lower. For the systems E-Mail and Elevator we counted four unique values. The systems AJstats,

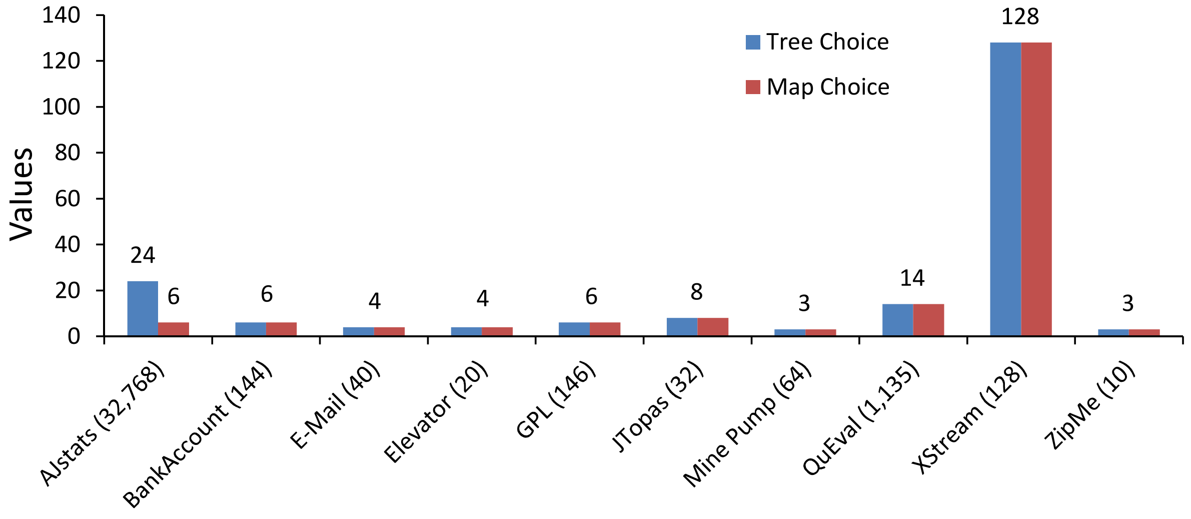

Figure 5.9: Maximal size (i.e., number of concrete values) of conditional values. The number for map choice also defines the maximal number of unique values in choices.

BankAccount, and GPL have all six unique values. For the system AJstats, VarexJ created a tree choice with 24 values, because we only implemented simplification over one level of the tree. The system QuEval already creates choices value with 14 unique values. The highest interaction has the XStream with 128 unique values what is already equal to the number of configurations. In XStream all features modify the same string value, thus the string has 2^n different results. That XStream has such a high degree of interaction might be the reason that map choice is faster for this system. The map choice has only one node which contains all values. In contrast, the tree choice contains of n - 1 nodes for n values (i.e., leafs), when there are no duplicates.

The sizes of choices we measured only represent the highest degree of interactions. Because the effort to execute the system depends on how oft such values are used for calculations, a distribution of the size of conditional values would be required. Future work should measure this distribution to identify eventual bottlenecks for variability-aware execution specific for the system, to reduce the analysis effort or to design optimized test cases.

5.5 Discussion

We showed that a variability-aware interpreter can reduce the effort for testing by orders of magnitude. Only for the system ZipMe variability-aware execute took more time than a brute force approach. However, the system only contains 10 configurations, thus the effort for calculation with conditional values is higher than the savings with sharing of executions. Furthermore, VarexJ is often even faster than τ-wise testing with JPF for $\tau = 2$ and $\tau = 3$.

The overhead of execution with JPF compared to execution with JVM is immense, but also this overhead can be compensated with sharing of instructions. For the systems AJstats, MinePump, and QuEval VarexJ is faster than brute-force execution with the host JVM. The times for JVM can be much higher, because the systems we used are relatively small, and optimization mechanisms, such as just-in-time compilation have negative affects when the system is always executed within a new JVM. On the other hand we did not measure the effort for initializing the JVMs, as each configuration has to be initialized separately, and thus highly increase the time for executions.

Variability-aware execution saves values for all configurations. Thus, the memory consumption of VarexJ is up to 19 times higher than executions of only one system. We think that variability-aware execution not necessarily requires that much additional memory, because values can be shared and the number of unique values in not always that high. We think that the current garbage collection cannot handle conditional values and that unnecessary values are not released. Future work has to figure out what exactly causes these memory leaks in VarexJ and needs to lift the garbage collection to handle conditional values.

Comparison with JPF-BDD

We used several systems from the evaluation with JPF-BDD [Apel et al., 2013d]. They used model checking to simulate random inputs for E-Mail, Elevator and Mine Pump, what in not possible with VarexJ. For the system ZipMe, it is not clear what program parameters they used for their measurements. Their measurements outperform product-based executions. For the system AJstats, they compared variability-aware execution with execution of only 200 products. When comparing extrapolated results, then the execution with VarexJ is around twice as fast as JPF-BDD. In contrast to the measurements of with JPF-BDD, we did not measure the time for initialization of the applications. To compare both implementations for variability-aware execution, both tools would need to be executed on the same hardware systems with equivalent measurements for time. The implementation of JPF-BDD improves the abilities of JPF to share instructions for features. Thus, the overhead of calculation with conditional values and the reasoning about configurations is not as high as for VarexJ. For systems with larger configuration spaces the sharing of VarexJ seems to outperform JPF-BDD.

Comparison with Shared Execution

We evaluated VarexJ on several programs from the evaluation of shared execution [Kim et al., 2012]. The evaluation compared variability-aware execution with brute-force execution with JPF. The evaluation contained measurements for time as well as measurements for executed instructions (again with program initializations). For the system GPL (Random 1) shared execution saved 94.9% of instructions and thereby saved 35% of execution time. In comparison VarexJ saved 95.6% of instructions and 76.6% of time. For XStream (0 Common, 30 Variable) shared execution is only able to reduce the instructions by 81% and only saved 11% of time. In contrast, VarexJ saved 96.9%

of instruction and thereby 55.78% of time. For the system JTopas (Many comments 2) the results for shared-execute were even worse with only 59% of saved instructions and an overhead of 81.4% for execution time. In comparison, VarexJ is able to reduce the instructions be 88.5% and save 80.9% of time. Because VarexJ can calculate with conditional values directly, it does not need to split executions when values differ among configurations. VarexJ can share executions more efficient than shared execution, and thus, saves redundant calculations and time. Again a direct and more meaningful comparison of both approaches requires that both tools are executed on the same systems with equivalent measurements. Furthermore, we were not able to compare our approach with details on shared execution, because the authors were not available despite requests.

Threats to Validity

For the measurement of execution times and executed instructions, we used a metric that is *comparable* among product-based execution with JVM and JPF, and variability-aware execution with VarexJ. Each execution requires an initial phase for the virtual machines and the application (e.g., class loading) that would add a static value to all executions. Because such initializations do not contain any variability, they would add an exponential overhead to the product-based measurements. For a fair comparison of the executions, we only measured the actual execution of the programs that have potentials for sharing. To minimize the computation bias, we did each execution 10 times and used the average execution time. The comparison of VarexJ with execution of one product is based on the average time. The times for execution can highly differ among configurations. However, to use average times is useful, especially because sampling algorithms try to cover several very different configurations.

To check the *validity* of our implementation and whether the variability-aware execution with VarexJ is equivalent to execution of single configurations, we compared the traces of the executions. For the product-based approach, we logged each executed instruction. Furthermore, we logged when a field is set with a new value. We additionally logged the corresponding context for variability-aware execution. Then we compared that the logs of execution of all configurations match the log of variability-aware execution (i.e., we compared the traces similar to the example shown in Table 3.1). With this comparison we found several fault during development and are now very confident that the implementation is valid at least for the evaluated programs. Furthermore, we are also confident that the measurements of instructions do not contain any bias.

We used the system QuEval to evaluate a system that is not used before in the context of variability-aware execution, and that is not developed as product-line. For QuEval variability-aware execution only scales for small program parameters because of the memory leaks. To implement variability-aware garbage collection was not possible at this point and should be part of future work.

Initially, we wanted to show the *scalability* of our approach to large real-world systems as in the evaluations of Varex [Nguyen et al., 2014]. Due to the unexpected difficulties with missing peer method this was not possible in our limited time. Because we needed to lift or introduce some peer methods for the systems we used, we are confident that automated lifting of peer methods to call native implementations is possible, and thus also variability-aware execution of larger systems.

A goal of this thesis is to *lift a complete language* to execute arbitrary programs. We lifted all 183 of JPF and applied VarexJ to 10 configurable systems of different size to show that VarexJ is able to be applied to a variety of programs. We only used small systems, thus a general conclusion to real-world systems is not possible. On the other hand, we provide a tool and an approach that gives better insights to the feature interactions and their causes in programs.

To evaluate *sharing* of computations, we counted the number of included features in the context of each instruction. This number indicates the size of the corresponding configuration space only indirectly. For example, the contexts $A \vee B$ and $A \wedge B$ contain the same number of features, but $A \wedge B$ is only a subset of $A \vee B$. To correctly measure the correlation between configuration space and instruction, the number of configurations which fulfill the context would need to be calculated for each instruction.

For RQ2, we also evaluated the *interactions* occurring in our systems. A good evaluation on feature interactions would require a mapping of variables and fields to the degree of interactions over time. Because such evaluation collects high amounts of data, and requires observing all values at any time, we only measured the highest degrees. The highest degree and the size of unique values for the value have only small expressiveness, because such interactions might only occur once and have minimal effects. On the other hand, small maximal degrees of interactions indicate that the overall system only contains small iterations. To really analyze the interactions in a system better measurements are required, which could lead to visualizations such as heat maps that can help to localize faults and indicate design failures.

5.6 Summary

In this chapter, we evaluated variability-aware execution with VarexJ on 10 configurable systems. At first, we discussed our evaluation framework to measure comparable results for the host JVM, JPF, and VarexJ. Then, we showed that variability-aware execution can reduce the number of required instructions by orders of magnitude, and thus also reduce the time compared to product-based executions. Our measurements showed that variability-aware execution does not require exponential memory compared to execution of one configuration. Furthermore, our evaluation indicates that instructions are likely to be executed on large configuration spaces where only some features are defined. To analyze interaction in systems, we analyzed the number of features in conditional values and the size of conditional values. Both numbers are relatively high and indicate that some interactions are hard to detect with τ-wise interaction testing. Finally, we concluded our results, compared our evaluation with previous variability-aware approaches, and discussed threats to validity.

6. Related Work

Research on analysis of configurable software and software product lines was active in the last decade. New approaches have been proposed to handle the challenges that come with variability, such as the configuration-space explosion. Because our work focuses on testing and execution of configurable systems we refer to recent surveys on analysis of software product lines [Thüm et al., 2014a] and tools thereof [Meinicke et al., 2014].

Variability-Aware Testing

Several approaches for variability-aware execution were proposed in the last years. Similar to VarexJ, they take variability into account to share execution, and thus reduce effort for redundant calculations.

VarexJ is based on previous work on variability-aware interpreters for a WHILE language and for PHP. The interpreter for WHILE is only a proof of concept and is written for a toy language [Kästner et al., 2012b]. The PHP interpreter Varex is written for the plug-in application WorldPress [Nguyen et al., 2014]. The interpreter shows that the approach scales for 50 plug-ins on a real-world application, but the interpreter is only able to execute WorldPress. To execute further applications the interpreter requires further effort for lifting. With VarexJ, we developed a variability-aware interpreter for a complete language, namely Java Bytecode, that is shown to be able to execute a variety of different programs.

Kim et al. [2012] developed variability-aware execution based on JPF called shared execution. The approach was applied to three product lines with up to 146 configurations. The approach improves sharing among configurations, but the reduced number of instructions is not always good enough to outperform brute-force execution with JPF. Kim et al. [2012] reported savings for execution times of up to 53%. Our approach has higher potential of sharing than shared execution, and thus is more efficient. We evaluated VarexJ to the same programs as for the evaluation of shared execution and measured better performances for sharing of instructions and for execution time.

Austin and Flanagan [2012] proposed multiple facets (a kind of variability-aware execution) that uses faceted values (similar to conditional values) for information flow analysis based on a JavaScript interpreter. Instead of configuration options, they use access rights that can mark elements as private. They showed that their approach outperforms brute-force executions for already one feature and concurrent execution on four cores for four optional features. The approach was only evaluated on a 300 lines md5-crypto algorithm with up to eight features. Thus, the evaluation cannot be used for a general conclusion about the scalability of their approach to larger programs with more code and more configuration options. In contrast, our evaluation covered programs with up to 9,000 lines of code and 32,768 configurations.

Efficient variability-aware model checking with JPF is done with the extension JPF-BDD [von Rhein et al., 2011; Apel et al., 2011a, 2013d; Kästner et al., 2012b]. JPF-BDD uses BDDs to reason about feature combinations to efficiently join and share executions. Because JPF uses Java Bytecode instructions as transitions between states, the difference to execution is minimal. They reuse the abilities for sharing and multiple values of JPF and showed speed-ups compared to a brute-force execution by orders of magnitude. Due to scalability reasons and restrictions of JPF they were only able to evaluate JPF-BDD on small programs, similar to our evaluation. We applied VarexJ to the same programs as for the evaluation of JPF-BDD and gained similar results. For a meaningful comparison of both approaches, both tools have to be executed within the same environment. As VarexJ has a higher potential for sharing than JPF-BDD, we expect higher speed-ups for our approach on larger programs with more features. Thüm et al. [2014c] use JPF-BDD for variability-aware execution in combination with runtime-assertion checking of feature-based specifications. Model checkers are designed to handle variability. Thus, there are several more approaches for variability-aware model-checking that include feature dependencies into the verification process [Lauenroth et al., 2009; Classen et al., 2010; Asirelli et al., 2011; Classen et al., 2011]. These approaches only operate on models of the system, and thus cannot be used for testing.

With symbolic evaluation values can be treated as unknown symbolic values, similar to annotated fields in VarexJ. Reisner et al. [2010] use symbolic evaluation for variability-aware execution of configurable systems with runtime options. In contrast to a variability-aware interpreter only marked values can be handled symbolic. Thus, the approach cannot share executions as efficient as VarexJ and cannot join execution paths after the executions are split for different configurations.

Rozzle is a virtual machine for multi-execution of JavaScript to detect malware [Kolbitsch et al., 2012]. Similar to VarexJ, the tool can execute multiple execution paths within a single run using symbolic values. In contrast to conditional values that represent concrete values, Rozzle uses symbolic values that can lead to infeasible executions.

Sumner et al. [2011] proposed coalescing execution that uses vectors which represent multi-values as program input. Such a vector represents multiple concrete alternative values. In contrast to configuration options, these vectors directly introduce interactions, thus each time this multi-value is used for calculations the instructions have to be applied to all its values. On the other hand, they do not need to reason about feature selections. Sumner et al. [2011] reported an average speed-up by the factor of 2.3 for a multi-value with 30 entries.

There are several more variability-aware approaches for efficient analysis of configurable software systems, such as for type checking [Kästner et al., 2012a], static analysis [Bodden et al., 2013], and deductive verification [Thüm et al., 2012]. Similar to a variability-aware interpreter, these approaches are able to reason about variability, to reduce redundant calculations.

Product-Based Testing

Product-based testing is a common strategy to test configurable systems [Thüm et al., 2014a]. With product-based testing only one configuration is tested at a time, thus a specialized tool support is unnecessary. Because testing all configurations is often inefficient and impractical, product-based strategies try to reduce the number of configurations and test cases that need to be execution, while the effectiveness to find defects does not change.

A popular method to detect defects that are caused by feature interactions is combinatorial interaction testing [Cohen et al., 1997, 2007]. Combinatorial interaction testing creates and tests configurations such that all combinations of τ features are covered. With this strategy all defects caused by interactions of τ features can be detected while the number of configurations to test can be reduced drastically. Yet, the approach does not scale for higher τ than 5, because the generation of configurations gets expensive. Thus, detection of interactions of many features is impractical with combinatorial interaction testing. Perrouin et al. [2010] provides automated test-case generation satisfying τ-way coverage. Because not all feature combinations affect the programs execution Schroeder and Korel [2000] use input-output analysis to additionally reduce the number of configurations without reducing the fault detection rate. τ-way interaction testing only looks at the variability of the program, but not at its implementation. Because many defects might already be detected if the corresponding code would be executed, [Tartler et al., 2012] propose configuration coverage that creates configurations where each code fragment is active at least once. Our evaluation showed that variability-aware execution can be faster than some product-based approaches, such as pairwise testing.

Other strategies try to avoid redundancies in executions but keep the detection rates as if all products were tested. With automated test-case generation [Cichos et al., 2011; Lochau et al., 2012], the number of tests to execute and the number of products to test can be reduced. Because a test cases might not require all variability or the execution of the test is equivalent among several configurations, the number of configurations to test [Cabral et al., 2010; Kim et al., 2010, 2011, 2013] and the number of redundant test cases [Qu et al., 2011, 2012; Shi et al., 2012] can be reduced. The approaches can be very efficient, especially when only minimal variability is involved into the execution of the test case (e.g., for unit tests). When all features are involved, these approaches might be equivalent to brute-force testing of all configurations. For VarexJ, such analyses are unnecessary because only the features that are used are considered during execution.

7. Conclusion ana Future Work

In this thesis, we presented how testing can be efficiently applied to configurable Java programs. We used the approach of a variability-aware interpreter [Kästner et al., 2012b; Nguyen et al., 2014] to lift the virtual machine of JPF. Our JVM called VarexJ is able to efficiently execute all configurations simultaneously by sharing executions with an internal representation of variability.

To completely lift an interpreter requires significantly more effort than to reuse existing tools that are already able to handle variability, such as model checkers [Apel et al., 2013d] and theorem provers [Thüm et al., 2012]. However, a variability-aware interpreter has several advantages. The interpreter optimizes sharing of executions, and thus improves the efficiency of variability-aware execution. Furthermore, the interpreter gives better and easier insight into feature interactions because all variables can be observed anytime during program execution. With VarexJ, we showed that lifting of a complete language is possible in the limited amount of time of this thesis. Except of native methods, which require individual lifting, all elements of VarexJ are variability-aware, such as the method frame and bytecode instructions. Through the development of VarexJ, we gained experiences with lifting of applications using conditional values that can also be applied to programs beyond interpreters (e.g., model checker or static analysis tools). We discovered several refactorings that occur repeatedly and that could possibly be automated in future work.

VarexJ optimizes sharing for efficient variability-aware execution, but we also discovered new potentials to share further executions, such as for redundant method calls among different configurations. We showed that the term of redundant executions is not always clear, because sharing of executions can range from single instructions up to sharing of methods that are called on distinct paths or on several objects. Future work on sharing should evaluate the proposed improvements for variability-aware execution, and for other variability-aware analyses, such as model checking.

To evaluate our implementations, we applied VarexJ to 10 configurable programs. We compared the efficiency of VarexJ to the unchanged interpreter of JPF. Because our approach required this additional layer of JPF, but for product-based execution a standard JVM is sufficient, we also compared VarexJ to the JVM from Oracle. We showed that execution of all configurations with VarexJ can be faster than pairwise testing with JPF. Depending on the size of the configuration space, VarexJ was also able to outperform the host JVM. Because of unexpected difficulties with native methods, we were not able to apply VarexJ to larger real-world programs, but the results for the smaller systems seem promising.

With VarexJ, we developed a tool that gives detailed insights into the feature interactions of a configurable program. We evaluated the interactions occurring in our programs and found that high interactions of more than three features are common. The program XStream even contains an interaction among all seven features that results in 2^7 unique values.

The approach of a variability-aware interpreter not only improves the efficiency of testing configurable software. It also provides a tool that helps to analyze and understand feature interactions, how they occur and how they can be avoided. With these new insights, the ways of developing configurable systems can be improved, to provide better and less error prone software.

Future Work

We already discussed several improvements of our current implementation in Chapter 4 and Chapter 5. There are currently two major pitfalls of variability-aware execution with VarexJ that should be solved to execute larger real-world applications. The first is that not all programs can be executed because the support of native methods is incomplete. Future work should investigate how *native methods* can be lifted efficiently, eventually leading to an automated lifting. The second challenge comes with the immense *memory consumption* of VarexJ compared to product-based executions. We believe that variability-aware execution not necessarily requires as much memory as we measured, and that the current garbage collector cannot handle conditional values. Thus, unnecessary values are not released. Kim et al. [2012] discussed this problem for shared execution and provided some solutions that might be useful for VarexJ.

JPF itself is a Java program, which requires an additional JVM to be executed. This overhead increases memory consumption as well as the time for executions. We used JPF for the basis of VarexJ, especially because of the simplicity of JPF. To provide a more efficient variability-aware interpreter, virtual machines that do not require a second JVM to be executed could be used, but they might require higher effort for lifting. On the other hand, these JVMs might integrate native methods in a larger scope than JPF.

Execution of native methods is different from execution of Java Bytecode. Because we have no access to the variables used by native implementations, these methods usually cannot be lifted directly and need to be executed many times. This approach might usually work, but only for methods without side effects. These side effects can be fields of a native class as well as a file that is written. Future work needs to investigate how these challenges with side effects can be solved to provide sound variability-aware executions.

Variability-aware execution provides mechanisms to analyze feature interactions efficiently. With a variability-aware interpreter interactions of features can be observed during execution. These interactions can be visualized with heat maps that show the degree of interactions (e.g., on each value, or method). Furthermore, code coverage among all configurations can be calculated efficiently with variability-aware execution. This variability-aware code coverage can detect uncovered code or possibly dead code due to invalid feature combinations.

Bibliography

Bowen Alpern, Steve Augart, Stephen M Blackburn, Maria Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, et al. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44(2):399–417, 2005. (cited on Page 30 and 31)

Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. (cited on Page 14)

Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of Feature Interactions Using Feature-Aware Verification. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 372–375. IEEE, 2011a. (cited on Page 80)

Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Feature-Aware Verification. Technical Report MIP-1105, University of Passau, Germany, 2011b. (cited on Page 56 and 60)

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013a. (cited on Page 1, 13, and 15)

Sven Apel, Christian Kästner, and Christian Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering (TSE)*, 39(1):63–79, 2013b. (cited on Page 60)

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. ACM, 2013c. (cited on Page 2)

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, 2013d. (cited on Page 3, 37, 59, 60, 61, 74, 76, 80, and 83)

Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, and Stefania Gnesi. A Model-Checking Tool for Families of Services. In *Proceedings of the International*

Conference on Formal Techniques for Distributed Systems (FMOODS/FORTE), pages 44–58. Springer, 2011. (cited on Page 80)

Thomas H Austin and Cormac Flanagan. Multiple Facets for Dynamic Information Flow. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. ACM, 2012. (cited on Page 24 and 79)

Bernhard Beckert, Reiner Hähnle, and Peter Schmitt. *Verification of Object-Oriented Software: The KeY Approach*. Springer, 2007. (cited on Page 60)

Jonathan Bell and Gail E Kaiser. VMVM: Unit Test Virtualization for Java. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 576–579. ACM, 2014. (cited on Page 64)

Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 355–364. ACM, 2013. (cited on Page 81)

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 81)

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, 2003. (cited on Page 2)

Harald Cichos, Sebastian Oster, Malte Lochau, and Andy Schürr. Model-Based Coverage-Driven Test Suite Generation for Software Product Lines. In *Model Driven Engineering Languages and Systems*, pages 425–439. Springer, 2011. (cited on Page 81)

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999. (cited on Page 54)

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. ACM, 2010. (cited on Page 80)

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. ACM, 2011. (cited on Page 80)

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

David M. Cohen, Siddhartha R. Dalal, Michael L Fredman, and Gardner C. Patton. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Transactions on Software Engineering (TSE)*, 23(7):437–444, 1997. (cited on Page 2, 13, and 81)

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 2 and 81)

Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Understanding Linux Feature Distribution. In *Proceedings of the Workshop on Modularity in Systems Software (MISS)*, pages 15–20. ACM, 2012. (cited on Page 1)

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, 2011. (cited on Page 5 and 7)

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. Springer, 2013. (cited on Page 8 and 9)

Robert J. Hall. Fundamental Nonmodularity in Electronic Mail. *Automated Software Engineering (ASE)*, 12(1):41–79, 2005. ISSN 0928-8910. (cited on Page 60)

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

Marc R Hoffmann. Eclemma-Java Code Coverage for Eclipse, 2009. (cited on Page 60)

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. ACM, 2012. (cited on Page 66)

JTopas. JTopas: Java tokenizer and parser tools. Website, 2014. Available online at http://jtopas.sourceforge.net/jtopas/; visited on October 16th, 2014. (cited on Page 60)

Josh Juneau, Jim Baker, Frank Wierzbicki, Leo Munoz Soto, and Victor Ng. *The Definitive Guide to Jython*. Springer, 2010. (cited on Page 19)

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 6)

Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824. ACM, 2011. (cited on Page 7, 30, and 48)

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. ACM, 2012a. (cited on Page 7, 30, 48, and 81)

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. ACM, 2012b. (cited on Page 3, 4, 17, 18, 24, 30, 31, 43, 61, 79, 80, and 83)

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242. Springer, 1997. (cited on Page 1)

Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. Reducing Configurations to Monitor in a Software Product Line. In *Proceedings of the International Conference on Runtime Verification (RV)*, pages 285–299. Springer, 2010. (cited on Page 81)

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. ACM, 2011. (cited on Page 81)

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. IEEE, 2012. (cited on Page 3, 17, 24, 31, 37, 59, 60, 61, 70, 76, 79, and 84)

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 81)

Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-Cloaking Internet Malware. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 443–457. IEEE, 2012. (cited on Page 80)

Jeff Kramer, J Magee, M Sloman, and A 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 60)

D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo Jr. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering (TSE)*, 30(6):418–421, 2004. (cited on Page 2)

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. IEEE, 2009. (cited on Page 80)

Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 105–114. IEEE, 2010. (cited on Page 2)

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. ACM, 2013. (cited on Page 1, 3, 7, 30, and 48)

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Pearson Education, 2014. (cited on Page 21, 25, 38, 45, and 47)

Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental Model-based Testing of Delta-oriented Software Product Lines. In *Proceedings of the International Conference on Tests and Proofs (TAP)*, pages 67–82. Springer, 2012. (cited on Page 81)

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. Springer, 2001. (cited on Page 60)

Jens Meinicke. JML-Based Verification for Feature-Oriented Programming. Bachelor's thesis, University of Magdeburg, Germany, 2013. (cited on Page 31 and 56)

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)*. ACM, 2014. (cited on Page 79)

Bertrand Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, 1992. (cited on Page 60)

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. ACM, 2014. (cited on Page 3, 4, 17, 18, 24, 30, 57, 71, 77, 79, and 83)

Armstrong Nhlabatsi, Robin Laney, and Bashar Nuseibeh. Feature Interaction: The Security Threat From Within Software Systems. *Progress in Informatics*, 5:75–89, 2008. (cited on Page 2)

Changhai Nie and Hareton Leung. A Survey of Combinatorial Testing. *ACM Computing Surveys*, 43(2):11:1–11:29, 2011. (cited on Page 1, 13, and 15)

Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 459–468. IEEE, 2010. (cited on Page 81)

Justyna Petke, Shin Yoo, Myra B Cohen, and Mark Harman. Efficiency and Early Fault Detection with Lower and Higher Strength Combinatorial Interaction Testing. In *Proceedings of the International Symposium Foundations of Software Engineering (FSE)*, pages 26–36. ACM, 2013. (cited on Page 2 and 74)

Malte Plath and Mark Ryan. Feature Integration Using a Feature Construct. *Science of Computer Programming (SCP)*, 41(1):53–84, 2001. (cited on Page 60)

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

Hendrik Post and Carsten Sinz. Configuration Lifting: Software Verification meets Software Configuration. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 347–350. IEEE, 2008. (cited on Page 56)

Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443. Springer, 1997. (cited on Page 1 and 60)

Xiao Qu, Mithun Acharya, and Brian Robinson. Impact Analysis of Configuration Changes for Test Case Selection. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 140–149. IEEE, 2011. (cited on Page 81)

Xiao Qu, Mithun Acharya, and Brian Robinson. Configuration selection using code change impact analysis for regression testing. In *Proceedings of the International*

Conference on Software Maintenance (ICSM), pages 129–138. IEEE, 2012. (cited on Page 81)

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 80)

Martin Schäler, Alexander Grebhahn, Reimar Schröter, Sandro Schulze, Veit Köppen, and Gunter Saake. QuEval: Beyond High-Dimensional Indexing à la Carte. *Proceedings of the VLDB Endowment*, 6(14):1654–1665, 2013. (cited on Page 60)

Patrick J. Schroeder and Bogdan Korel. Black-Box Test Reduction Using Input-Output Analysis. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 173–177. ACM, 2000. (cited on Page 81)

Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 270–284. Springer, 2012. (cited on Page 81)

Julio Sincero, Reinhard Tartler, Christoph Egger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Facing the Linux 8000 Feature Nightmare. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*. ACM, 2010. (cited on Page 2)

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 80 and 81)

Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, pages 47–60. ACM, 2011. (cited on Page 3)

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 2 and 81)

Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract Features in Feature Modeling. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 191–200. IEEE, 2011. (cited on Page 61)

Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. Family-Based Deductive Verification of Software Product Lines. In *Proceedings of the International*

Conference on Generative Programming and Component Engineering (GPCE), pages 11–20. ACM, 2012. (cited on Page 17, 56, 81, and 83)

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*, 47(1):6:1–6:45, 2014a. (cited on Page 1, 2, 3, 4, 15, 16, 17, 79, and 81)

Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming (SCP)*, 79(0):70–85, 2014b. (cited on Page 48 and 62)

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)*. ACM, 2014c. (cited on Page 31, 60, and 80)

Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003. (cited on Page 4 and 31)

Alexander von Rhein, Sven Apel, and Franco Raimondi. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java code. In *Proc. Java Pathfinder Workshop*, page 82, 2011. (cited on Page 80)

Alexander von Rhein, Sven Apel, Christian Kästner, Thomas Thüm, and Ina Schaefer. The PLA Model: On the Combination of Product-Line Analyses. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 14:1–14:8. ACM, 2013. (cited on Page 3)

Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. Variational Data Structures: Exploring Trade-Offs in Computing with Variability. In *ACM SIGPLAN Symposium on New Ideas in Programming and Reflections on Software*. ACM, 2014. (cited on Page 6, 7, and 11)

Joe Walnes. XStream: A Simple Library to Serialize Objects to XML and Back Again. Website, 2014. Available online at http://xstream.codehaus.org/; visited on October 16th, 2014. (cited on Page 60)

Christian Wimmer, Michael Haupt, Michael L Van De Vanter, Mick Jordan, Laurent Daynes, and Douglas Simon. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4): 30:1–30:24, 2013. (cited on Page 30 and 31)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den Dezember 3, 2014