University of Marburg
Department of Mathematics & Computer Science

Bachelor Thesis

Variability-Aware Interpretation

Author:
Jonas Pusch

October 11, 2012

Advisors:
Prof. Dr. Klaus Ostermann
University of Marburg
Department of Mathematics & Computer Science

Prof. Dr. Christian Kästner
Carnegie Mellon University
Institute of Software Research

Sebastian Erdweg, M.Sc.
University of Marburg
Department of Mathematics & Computer Science

Pusch, Jonas:
Variability-Aware Interpretation
Bachelor thesis, University of Marburg, 2012.

Contents

List of Figures

List of Listings

List of Abbreviations

SPL	software product line
AST	abstract syntax tree
AOP	aspect-oriented programming
FOP	feature-oriented programming

Chapter 1

Introduction

1.1 Overview

Software product lines (SPL) offer the ability to create multiple similar software products from a shared set of features. These features satisfy the needs of a certain domain or market segment and are connected to software components, in which they are implemented. The goal of software product lines is to decrease maintenance- and development times when distributing various similar software products, while simultaneously increasing the flexibility and quality of the created product variants. An advantage of SPL is the high level of reusability that enables developers to pick a specific subset of features and thereby get a custom-made product variant.

To create a finished product, features from a shared pool are selected and a generator connects software components according to the specific feature selection. Products with the same features, thus reuse the same existing software components, which do not have to be rewritten for every new product. Furthermore, changes to the implementation of a feature in the shared pool affect all products that use this feature and remove the need to carry out changes to the code of every single software product. Therefore, software product lines as a development paradigm, obtain an increasing relevance in economy. The ability to provide custom-made software, enables companies to extend their product range, while saving working hours helps to maximize profits.

An increasing relevance of software product lines, leads to the demand of testing a whole product line. Traditional approaches focus on testing all configured programs of a SPL. Considering n features in a product line would require to analyze up to 2^n configurations separately. This kind of approaches can be seen as a brute-force method of testing SPLs, because indeed every product that can be derived of the product line's common set of software components, gets analyzed. An obvious problem with this approach is redundancy. The fact that different products of an SPL reuse the same software components, implies a certain similarity concerning these derivates of an SPL. This results in similar parts being analyzed multiple times, which is not desirable. In addition, testing 2^n products is not viable from a certain number of features.

One alternative to the brute-force approach are sampling strategies [Cabral et al., 2010]. In this approach only a subset of all configurations is analyzed, selected randomly or by a specific criterion. Sampling strategies solve the problem of testing all generated products, but in return they cannot provide an exhaustive result for a whole product line. To still achieve this goal, researchers have recently drawn attention to variability-aware analysis of SPL. Performing variability-aware analysis means, that the variability of a product line is included in the analysis process. In contrast to other methods, variability-aware analysis are not performed on the products of a SPL, but rather on the not yet configured implementation of a software product line. Thus, only one analysis is performed, that takes variability into account when needed and redundant parts of the program are not analyzed again and again.

Variability-aware interpretation finally connects the process of interpreting source code with the principles of variability-aware analysis. In the above-mentioned traditional approaches, where tests are performed on configured products, no variability is left in the source code, because the feature selection has already been done. Interpreting source code in a variability-aware fashion, however, describes a different strategy. In this method, source code is interpreted together with the instructions that specify the feature context, in which code segments are executed. The output of the variability-aware interpretation process are results enriched with variability, which can subsequently be configured to get the concrete result for a specific feature selection. Benefits of variability-aware interpretation are the possibility to execute test cases in all configurations and the avoidance of redundant interpretation, as source code gets interpreted only once.

1.2 Contributions

In this thesis the concept of variability-aware interpretation is introduced as an alternative software product-line testing approach to existing brute-force or sampling strategies.

Initially, the structures used to represent variability are explained. This includes explanations about how variability is specified at source code level and how it is transferred to a representation, which can be used as input for variability-aware analysis. These informations provide basic knowledge for understanding the further steps towards variability-aware interpretation.

On top of that, the prototype of a variability-aware interpreter for a variability-enriched While language is introduced as the main contribution of this thesis. The interpreter is able to execute statements annotated with ifdef variability, syntactically similar to the C preprocessor's conditional compilation directives. Also it can execute test cases in all configurations of a product line by using assertions. The concept of how the interpreter reasons about variability is explained and its implementation is shown based on code examples. The code of the interpreter is available at `http://github.com/puschj/Variability-Aware-Interpreter`.

This work shows, that variability-aware interpretation can speedup software product-line testing in comparison to brute-force analysis by applying the principles of variability-aware analysis on interpreting source code. Therefore, both approaches are compared in an empirical evaluation, that features three benchmarks including favorable cases to show the interpreter's potential and a set of 100 random generated test product lines to provide a comprehensive result for a rich set of subjects.

1.3 Outline

Chapter 2 provides an overview of software product lines in general (Section 2.1) and how variability can be specified in programs (Section 2.2). Afterwards, testing of software product lines is described and the problems that occur thereby (Section 2.3). The chapter closes with describing variability-aware analysis as an approach to address problems concerned with product-line analysis (Section 2.4), as well as the structures needed to perform variability-aware analysis on programs.

The variability-aware interpreter is described in Chapter 3. First, this thesis declares a problem statement that specifies the goals associated with developing an interpreter (Section 3.1). After that, the underlying concept (Section 3.2) and the implementation (Section 3.3) of the interpreter itself are explained in detail.

In Chapter 4 an empirical evaluation of the variability-aware interpreter is performed. At first, it is explained which method has been used for the evaluation and why (Section 4.1). Then, the way of comparing the approaches under test is shown (Section 4.2), before the subjects of the evaluation are described (Section 4.3). The results of the evaluation are shown in Section 4.5. Finally, the results are discussed (Sec. 4.6) and possible threats to validity are mentioned (Sec. 4.6.1).

Future research targets concerning variability-aware interpretation are shown in Chapter 5 and related work is shown in Chapter 6. Lastly, this thesis gets concluded in Chapter 7.

Chapter 2

Variability in Programs

2.1 Software Product Lines

A *Software Product Line* [Kästner, 2011] is a set of similar software systems, with each one sharing a common set of features to satisfy the needs of a certain domain and each one being built from a common set of software components. A domain can be a certain market segment or the needs of a specific task. The demands of a domain are modeled as features, which in turn are used to specify concrete program variants. SPL development, however, differs noticeably from development of single products. It is divided into Application Engineering and Domain Engineering. The latter represents the main difference to traditional development. Instead of engineering one single product for a specific order, Domain Engineering involves collecting the needs of a whole domain, such as a market segment, and converting the demands of the field to reusable software artifacts, that can later be tailored together to form a final derivate of the product-line. Additionally, Domain Engineering includes finding appropriate means of how to reuse this artifacts when building a concrete product. Application Engineering on the other side, describes the process of developing a single product, that means one single derivate of a product-line. Unlike in traditional development, no new product gets implemented in this step, but rather the artifacts created in Domain Engineering are used to create the final product. The Application Engineering process involves selecting appropriate features for the requested product. These features represent a specific subset of the above-mentioned software artifacts, which afterwards get tailored together and form the final software product.

Figure 2.1 summarizes the process of product-line development. Domain- and Application Engineering are divided. In the Domain Engineering process, analyzing the domain and collecting domain knowledge firstly results in a feature model, from which specific features later can be selected to form a product. Secondly, a set of reusable software artifacts is implemented, with all components representing a specific feature of the domain. At Application Engineering level, the first step is to perform a feature selection that fits to the requirements of the requested product. Providing a specific generator for tailoring together specific software artifacts belongs also to Application Engineering. The generator finally connects Domain- and Application Engineering by taking inputs of both branches and constructing the final product. As input, it takes the set of reusable software artifacts, which have been implemented in the Domain Engineering process, together with the above-mentioned feature selection that tells the

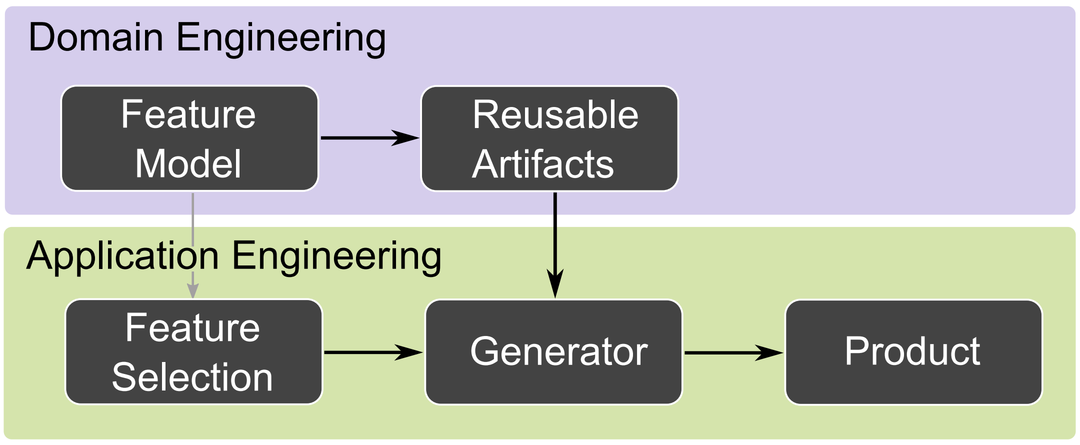


Figure 2.1: Product-line development process.

generator which artifacts are used in the construction process. The outcome, finally, is a derivate of the product-line, tailored to the customer's specific needs.

As already explained, the way of constructing programs in product-line development, differs clearly from traditional approaches. The division into Domain- and Application Engineering and the fact that this involves implementing software artifacts that satisfy the needs of a whole domain, do not imply product-line development as very cost-efficient at first glance. However, SPLs unfold their full potential, when many similar products have to be developed, which can profit from the high degree of reusability, that product-line development provides.

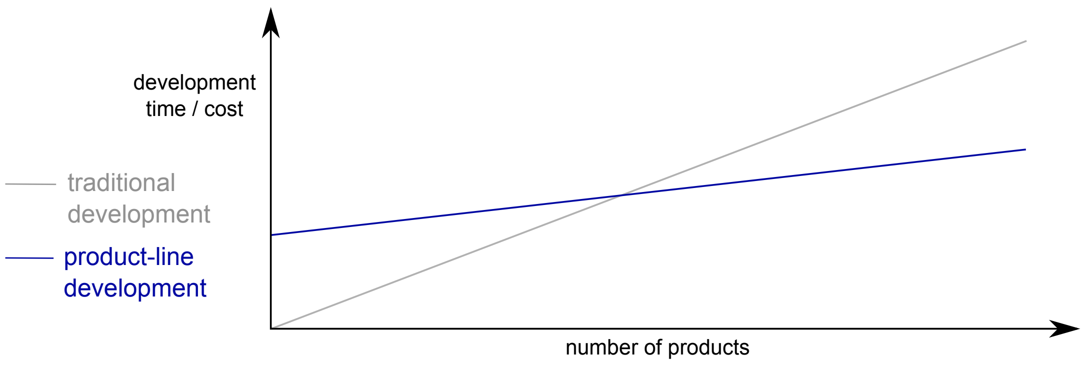


Figure 2.2: Time/cost-effectiveness of different development approaches.

The benefit of software-product-line development is illustrated in Figure 2.2. While developing a product-line initially produces higher costs, an increasing number of products that can be constructed using SPL reusability, results in a less rising cost-graph compared to traditional software development. The higher initial costs in product-line development are closely related to the Domain Engineering process. This phase, however, is executed only once and further products profit from already existing software components, that just have to be woven together. That way, it can be explained, why software-product-line development receives an increasing attention. At a certain number of products, every further product increases the benefit of just having to reuse existing components. The exact number, at which the advantage turns towards product-line development, depends on the size and properties of a product-line.

2.2 Variability in Product Lines

In Section 2.1, Domain Engineering was introduced as a process, where domain knowledge is used to create a set of reusable software artifacts, which can afterwards be tailored together by a generator according to a certain feature selection. Creating a set of reusable software artifacts requires a method to map specific code fragments to features. This is necessary for the generator to pick up the correct pieces of code when creating the final product. In the past, multiple ways of implementing variability in SPL have proven working. This work uses preprocessor directives because of their simplicity. The following list provides an overview of established ways of implementing variability that helps to bring the approach used in this work in line with other approaches of implementing variability.

- Runtime Variability
  This topic includes command-line parameters, configuration files and global configuration variables. Configurations are checked at runtime and there is no explicit generator, as unused functionality is still in the source code, even if not used due to runtime checks.

- Build Systems and Version Control Systems [Staples and Hill, 2004]
  When implementing variability with build systems or version control systems, program variants get mapped to build scripts in the case of build systems, or branches in the case of version control systems. Existing tools provide good tool support for this method of building SPL, but the downside on the other hand is, that developers operate on variants, not features in isolation. Merging different variants requires high efforts and arbitrary combinations of features are not possible.

- Aspect-Oriented Programming (AOP) [AsJ, Kiczales et al., 2001]
  Aspects in AOP are able to manipulate existing class hierarchies and intercept method calls or field access without having to change existing code. This so-called principle of obliviousness enables developers to add feature functionality as independent aspects to an existing code base. As aspects are one-to-one mapped to features, a product can be created by including only the selected features, and thus aspects, in the compilation process.

- Feature-Oriented Programming (FOP) [Batory et al., 2003, Kuhlemann et al., 2007]
  FOP also follows the principle of obliviousness, as an existing code base can be refined in feature modules. Refining a class, developers can add new methods or add code to existing methods. The idea is also to have a one-to-one mapping between features and feature modules. When creating a product, a special composition algorithm processes all refinements that were included by a certain feature selection. That way, the existing code base gets modified only by the refinements, which were chosen due to the feature selection.

- Preprocessors [Liebig et al., 2010]
  Using preprocessors, the source code is transformed before the compiler processes it. Developers specify code fragments that will either be left or removed according to a specific configuration, when the preprocessor is called on a source file. This

method guarantees, in contrast to Runtime Variability, that only the code of a specific variant gets delivered. Preprocessors do not separate the code that belongs to features from the common code base, which can lead to messy code, but they provide a easy-to-learn way to implement variability.

This work focuses on preprocessor directives, more specifically on *ifdef variability*, to specify feature contexts. This type of preprocessor directives receives its name because feature contexts are specified using #ifdef statements around variable code fragments.

Listing 2.1 shows an example of a variable assignment. Although there are other approaches to realize variability in programs (see above), ifdef variability offers a very simple and easy understandable way to specify feature contexts in source code. Additionally, using ifdef variability is quite common in product-line development, with the Linux kernel as its most prominent example. These are the reasons for focusing on the preprocessor approach in this work.

```
#ifdef A
x = 1;
#endif
```

Listing 2.1: Exemplary ifdef statement.

2.3 Testing Product Lines

Software testing is the process of examining the quality of a software product or a service. Tests often are conducted to prove that a certain program meets some predefined requirements or satisfies the customer's needs. A special method of software testing is *unit testing* [Olan, 2003]. Here, individual parts of the program (units) are tested separately. Unit tests are often created by programmers during the development process where they rely on unit testing frameworks, which are available for most of the common programming languages.

Concerning software product lines, testing is also important, because there are many different program variants, that can be derived from one set of software components and in the best case, the test result should provide an exhaustive output for the whole product line. However, testing a whole product line is different from testing a single program, because product lines include feature contexts next to the normal source code, which requires a more sophisticated approach.

The easiest way to conduct unit tests on product lines is to perform them on the products. In the Application Engineering process, the final outcome is a configured program, which contains a subset of the product line's shared set of software components, according to a specific feature selection. This product does not longer contain variability, because the generation process removes feature context declarations, after it has been checked whether their nested code block gets included in the final product, or not. For that reason, a default unit test can be executed on the generated product. The results of all test executions of all products that can be derived from a product line, provide an exhaustive test result for the product line as a whole.

When developing a product line, the number of features determines the possible number of different products, which can be derived. Another factor, that has influence on the maximum number of products, are dependencies between features. For example, when the selection of one feature implies that another feature already has been selected, the maximum number of derivates decreases. Assuming no dependencies between features, a product line with n optional features has 2^n possible products. This leads to huge numbers of products that have to be tested, when the number of features increases. Performing product-line analysis by testing all possible products, therefore is declined as a brute-force approach. High variability leads to high complexity when testing product lines. Considering a product line with 320 optional, independent features would lead to more possible products than the estimated number of atoms in the universe. Even with great computing power, testing a product line with about 10000 features (2^{10000} variants), like the Linux kernel, would not provide a result in a reasonable amount of time, when using the brute-force approach.

A possible solution for still getting a result, when testing a product line with such high numbers of features, is to reduce the number of products being tested. *Sampling strategies* [Cabral et al., 2010] focus on selecting some of the product-line variants for testing. Variants are selected randomly or by a specific criterion, with all criteria aiming at providing a more meaningful result by choosing the tested variants wisely. Though sampling strategies can provide a result, where the brute-force approach would not be able to provide a result in an acceptable amount of time, their result is not fully exhaustive. Even when choosing variants under test wisely, there is no guarantee for getting an expressive result, because still not all variants have been investigated. Therefore, sampling strategies do not provide an appropriate solution for getting exhaustive test results. They only solve performance issues when trading off performance against result quality, which is not desirable. In this work, the problem of getting an exhaustive result, while still not checking every possible variant, is addressed.

2.4 Variability-Aware Analysis

Because of the obvious performance issues presented in Section 2.3, researchers have drawn attention to an approach, where analysis are not performed on the products of a product line. Instead, they focus on a method, where analysis are conducted on the not yet configured software components of a product line, including variability in the analysis process. Analysis following this scheme are called *variability-aware analysis*, which shows their intent to handle variability while analyzing the product line. Existing examples for the use of variability-aware analysis are type checking [Kästner et al., 2011a], static analysis [Kreutzer, 2012] and model checking [Lauenroth et al., 2009]. This work omits further details concerning these use cases as they can be taken from the referenced publications. Variability-aware analysis address the above-mentioned performance issues in the way that product-line code is analyzed only once, instead of checking every possible product. Doing this, variability-aware analysis use the fact, that the variants of a product line normally have many similar code passages, that would be analyzed over and over again in the brute-force approach. Variability-aware analysis improve performance, because they analyze equal code only once and still take variability into account when needed. That way, additional effort is only needed for differing code passages.

Figure 2.3 illustrates the difference between variability-aware analysis and the brute-force approach, by showing two possible ways (red/green) of getting an analysis result for a product line. The red path shows the brute-force analysis, which initially

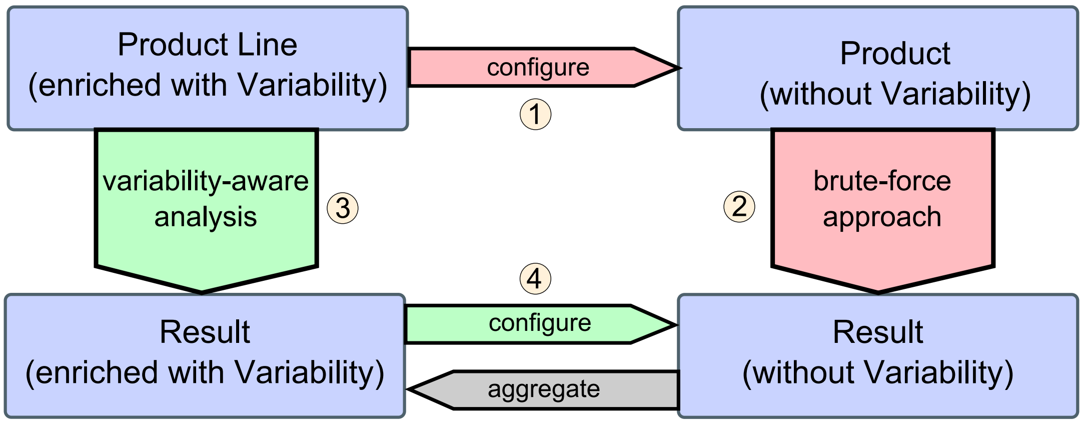


Figure 2.3: Variability-aware analysis in comparison to brute-force analysis.

configures all possible products (Step 1) and then conducts analysis on all of them (Step 2). On the contrary, the green path describes the variability-aware analysis, that is performed on the entire product line, instead of configured products (Step 3). The outcome of the analysis process is a result enriched with variability, because it has to store the value of all investigated properties for each configuration. Afterwards, the result for one specific configuration can be deduced from the variability-enriched result (Step 4, top). To get a result which describes the whole product line, it is also possible to aggregate all results from the brute-force analysis (Step 4, bottom). Though the results of both approaches should be equivalent, the variability-aware approach (green) is expected to be faster than the brute-force approach (red), because common code segments are analyzed only once.

2.4.1 Abstract syntax trees

When performing variability-aware analysis, a structural representation of code is needed. An abstract syntax tree (AST) offers this kind of structure, as it is a representation of the source code in tree form, based on the syntax of a certain programming language. When a specific construct of the programming language's syntax is used in the source code, it gets denoted as a tree node. Figure 2.4 shows an exemplary code fragment together with its corresponding AST. The root of the AST contains all top-level statements, that are in the source code: an assignment to variable x (Line 2) and an if statement (Line 4-6). The left branch decodes the assignment with an Assign node and has two children, Id and Num. Id nodes represent access to a variable and have one further child node for the name of the accessed variable (x in this case). The if construct is decoded in the right branch of the AST, with its subtrees for the condition and the statement that gets executed when the condition evaluates to true.

Abstract syntax trees form the basic structural representation of source code in this work. However, when developing product lines, the source code gets enriched with variability. This brings up the need for a structural representation of variability also, which is solved by extending the standard AST with constructs for variability. How variability is represented in AST is further explained in Section 2.4.2.

```
1 begin
2 x = 1;
3
4  if (x > 0) {
5    y = 1 + 2;
6 }
7 end
```

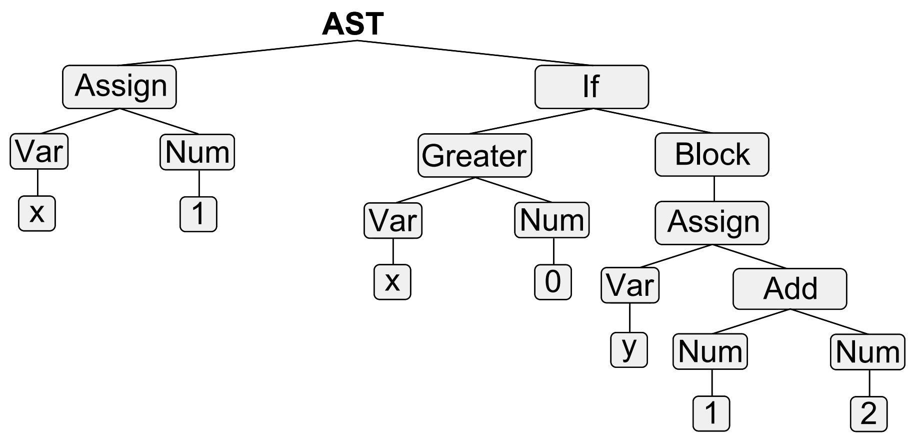


Figure 2.4: Exemplary code fragment with its corresponding AST.

2.4.2 Variability representation

When parsing product-line source code [Kästner et al., 2011b], the parsing process provides another kind of information next to the normal source code: the feature context, in which an instruction should be applied. Therefore, variability needs to be structurally represented when parsing variability-enriched code. Compile-time variability gets woven in abstract syntax trees in this work. More exactly, the feature context in which a node should be present, is also stored in the AST. This is done by wrapping nodes with a *presence condition*, a propositional formula over existing features, whose result determines the presence or absence of the node in the final product.

```
1  begin
2  #ifdef A
3  x = 1;
4  #endif
5
6   if (x > 0) {
7     y = 1 +
8       #ifdef B
9         2
10      #else
11        3
12      #endif
13    ;
14 }
15 end
```

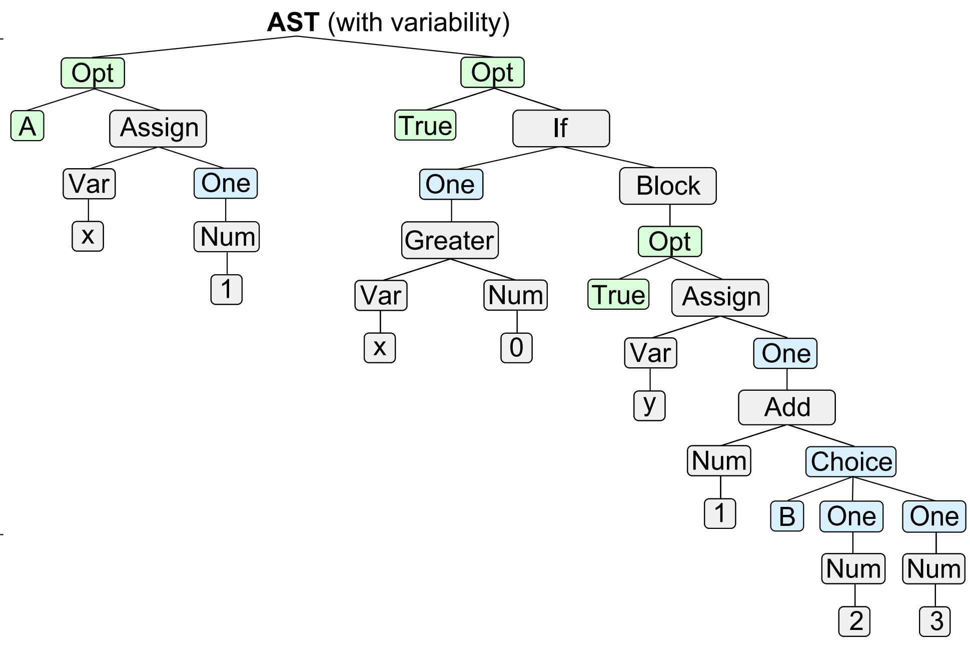


Figure 2.5: Variable code fragment and the corresponding AST with variability.

Figure 2.5 continues the example from the last section, but now introduces ifdef variability in the source code and variability nodes in the AST, respectively. Variability nodes can be optional (green) or conditional (blue). For example, the first assignment shall only be executed in variants with feature A selected (Lines 2-4). In the corresponding AST, the assignment is wrapped with an Opt node that also includes the feature context in which the Assign node later influences the execution (left branch).

Conditional nodes are used when different configurations yield different values, as with the assignment to y (Lines 7-13). In configurations where feature B is selected, the left side of the addition has the value 2, otherwise it yields 3, which is denoted using a Choice construct. On the contrary, when a conditional value is equal in all configurations, it is denoted using One, e.g. the right side of the assignment to x (Line 3).

After looking abstractly at the constructs to represent variability, the following paragraph describes how they are realized as constructs of an object-oriented programming language (Scala). Figure 2.6 shows, that there are two constructs available to handle variability: Opt[T] and Conditional[T]. Opt[T] is used for optional elements of type T, e.g. a list of statements with some of them only being present in specific configurations (List [Opt[Statement]]). Its two fields, feature and entry, specify the presence condition, that determines in which configurations the element is present on the one hand, and the element of type T itself, that is wrapped by the Opt[T] class on the other hand.

Presence conditions are of type FeatureExpr, which is a class especially designed for propositional formulas. FeatureExpr provides useful functions on propositional formulas, e.g. querying if the formula is a tautology or a contradiction, concatenating two formulas (*and*, *or*), negating the formula (*not*) or checking if a specific feature selection satisfies the formula. True indicates, that a node is present in every configuration.

While Opt[T] manages optional elements and mainly occurs in lists, Conditional[T] is the construct for elements, that have different values regarding to specific configurations. Lists of optional elements can have an arbitrary number of elements in different configurations, but conditional elements have exactly one value for each configuration. Conditional[T] is realized as an abstract class, that has two subclasses. One[T] simply is a container for elements of type T. Choice[T] is used to choose between two values depending on a presence condition. More specifically, if the presence condition evaluates to $true$, the value of this choice object for the current configuration is in the then-branch, otherwise it is in the else-branch. thenBranch and elseBranch are of type Conditional[T] again, which makes it possible to build nested choices. As a result, it can not only be chosen between two values, rather specifying multiple alternative elements of type T is possible.

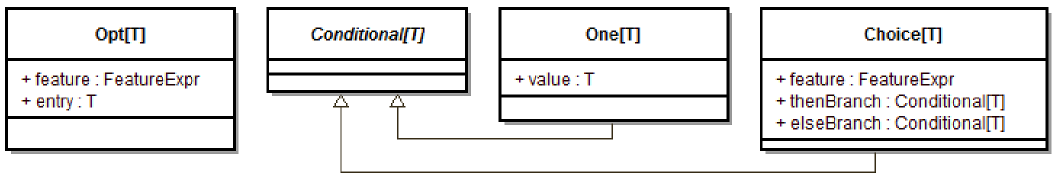


Figure 2.6: UML model of conditional classes.

2.5 TypeChef Variability-Aware Parser Framework

The TypeChef Variability-Aware Parser Framework [Kästner et al., 2011b] provides classes for parsing variability-enriched code as illustrated in Figure 2.5. This work uses the TypeChef parser as a block-box framework for creating ASTs with variability that can then be used in the variability-aware interpretation process. Concerning #ifdef variability, the framework contains a java parser that is capable of reading feature declarations inside of comments (//#ifdef and //#if), which is used out of the box. Creating ASTs with variability is done by using the parse methods provided by the framework that are capable of reading feature contexts. The following list briefly presents some of the possibilities, that are provided by the TypeChef parser:

- parse conditional elements
  Element can be parsed conditional, which means that it can be wrapped with an #ifdef statement for specifying its feature context.
- parse optional lists
  A list of optional elements can be parsed and the parser accepts #ifdef statements around each element of the list.
- parse sequences
  Parsing sequences of elements is used to parse compound constructs, such as a while loop, which contains a keyword, a condition and a body.
- map parse results
  Parsed elements can be mapped to a certain type. This is, for example, used to convert parsed tokens to custom AST nodes.

Further details on how exactly variability contexts are parsed using the TypeChef Variability-Aware Parser Framework have already been published in previous work and are no topic in this work. How using the framework is done programmatically, is illustrated in Section 3.3.

Chapter 3

Variability-Aware Interpreter

3.1 Problem Statement

In order to analyze whole product lines, a sophisticated approach is needed, that provides an exhaustive result in a reasonable amount of time. Right now, analyzing whole product lines is performed either with brute-force analysis, that require to analyze up to $2^{\#Features}$ configurations, or with sampling strategies, that reduce the number of tests, but cannot yield an exhaustive result for the whole product line. When focusing on analyzing configured products, the dependency between the number of features and the effort needed to receive a test result remains, thus providing a bad scaling for product lines with many features. Researchers therefore draw their attention to variability-aware analysis, a technique to perform analysis on the product-line code itself, taking variability into account and reducing test effort by analyzing shared code passages only once.

The above-mentioned performance problems for product lines with high numbers of features can be transferred to the interpretation of source code. When interpreting variability-enriched product-line code, developers want to be able to check a certain property, for example, with an assertion. According to brute-force analysis, this problem would be solved by interpreting every possible variant of the product line individually, which means also executing the assertion in every variant and thus getting an exhaustive result for the whole product line.

This method, however, leads to performance issues for product lines with many possible configurations, as has already been explained above. Therefore, the goal is to transfer the benefits of variability-aware analysis, which already provide strategies for reducing analysis effort in comparison to the brute-force approach, to the interpretation of source code. The resulting approach, *variability-aware interpretation*, takes varability into account when interpreting source code and as with variability-aware analysis, source code is interpreted only once and a speedup is received by not interpreting common code passages again and again. To perform variability-aware interpretation on source code, an interpreter must be implemented, that supports reasoning about variability in the manner of variability-aware analysis.

This work proposes to research the practicability and performance of such a variability-aware interpreter that accepts a While language and that applies the principles of variability-aware analysis to interpreting source code.

3.2 Concept

This section describes the concept of a variability-aware interpreter, using the principles of variability-aware analysis to perform testing on product lines as a whole. Interpreters are programs that execute code of a certain programming language at runtime. Traditional interpreters operate on ASTs (Sec. 2.4.1), which are the representation of a certain code fragment. Additionally, they use a store, where the values of all used variables are stored.

On executing a code fragment, the interpreter reads from the store, when variables are accessed and writes to the store, when variables are assigned, thus returning an updated store at the end of the execution. This process could be described with the following function: execute(s: Statement, store: Store): Store. However, when interpreting source code in a variability-aware fashion, the components used in the interpretation process differ. As shown in Section 2.4.2, ASTs are now enriched with variability. The interpreter also must be aware of the variability context it is currently operating in, so this variability context is another input for the variability-aware approach. Finally, the variability-aware interpreter stores its variables also in a variability-aware store, that maps variable names to conditional values, instead of just plain values. Using function notation, the variability-aware interpretation would be described with: execute(s: Conditional[Statement], store: VAStore): VAStore.

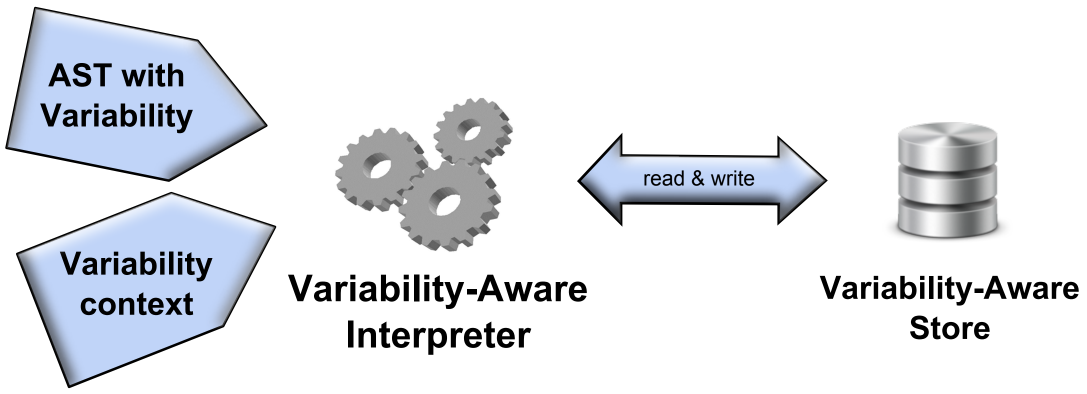


Figure 3.1: Architecture of the variability-aware interpreter.

Figure 3.1 describes the architecture of the variability-aware interpreter. Next to a different AST and Store, which are adjusted to support variability, the main difference to traditional interpreters is the use of a variability context in the interpretation process. On executing instructions it enables the variability-aware interpreter to be aware of its current variability context.

Variability contexts are propositional formulas, that describe the configurations in which the current code fragment gets executed. For example, an assignment to variable x within a #ifdef $A \wedge B$ block, will result in the interpreter executing the assignment with variability context $A \wedge B$. This means that the value of the assignment is only being stored for configurations, where feature A and feature B are selected. To be able to save variables depending on a certain configuration, the store used in variability-aware interpretation is not a mapping from variable names to values (Map[String, Value]) as in traditional interpretation, rather it maps variable names to conditional values (Map[String, Conditional[Value]]).

One important thing that speeds up variability-aware interpretation in comparison to brute-force analysis, is the *locality* of variability. Looking at the variability-aware store, variability is stored as local as possible. A less local implementation of a variability-aware store would be a conditional mapping of names to values (Conditional[Map[String, Value]]), which causes the problem, that a whole map must be stored for each configuration. However, keeping variability local, provides the ability to exploit *sharing*, which means, that structures are used for all configurations. When a value changes in a single configuration, a store of type Conditional[Map[String, Value]] would create a new modified Map for that configuration, thus redundantly storing all values that remain the same. If afterwards, a value changes in all configurations, the update has to be executed on every existing Map. By storing variability as local as possible (Map[String, Conditional[Value]]), updates on values affect only the value itself, because variability is stored at value level.

How values are stored in a variability-aware fashion on executing statements is shown in Figure 3.2. While the assignments to x have no variability context, the first assignment to y is executed in a specific variability context. In configurations where feature A is selected, the value 2 is assigned to y (Line 4) and when feature A is not selected, the value of y is 3 (Line 6). Below the code fragment, the state of the store after the variability-aware interpretation process is illustrated. While the value for variable x is unique and thus the same in all configurations, the value for y is conditional.

Programmatically, the final values are represented as One(2) and Choice(A, One(3), One(4)) as described in Section 2.4.2. Figure 3.2 also shows, how sharing can reduce analysis effort in comparison to the brute-force approach. When x gets incremented in Line 8, again with no specific variability context, this requires only one execution, because x not yet depends on a certain configuration, which means that it is *shared* between configurations. Looking at brute-force analysis, the assignment would be executed twice, because one selectable feature yields two possible products, each with one incrementation. Incrementing y on the other hand (Line 9), requires the variability-aware interpretation process to update y's value for all stored configurations, which are in this case A and $\neg A$, resulting in both approaches requiring two instructions. In other words, only code passages with variability that affect the analysis result cause additional effort during the variability-aware interpretation process. Code passages that are shared between configurations, therefore make up the difference that speeds up variability-aware interpretation compared to checking all configurations individually.

```
1  begin
2  x = 1;
3  //#ifdef A
4  y = 2;
5  //#else
6  y = 3;
7  //#endif
8  x = x + 1;
9  y = y + 1;
10 end
```

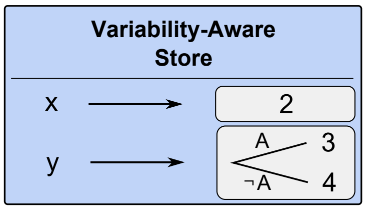


Figure 3.2: Storing conditional values.

Evaluating expressions. Another functionality of the variability-aware interpreter, which has also been used in the above example, is the evaluation of expressions. Conceptually, the interpreter takes a conditional expression and returns a conditional value. In this case, every alternative expression must be evaluated separately in its own feature context. For example, when variable y is incremented (Line 9), the interpreter has to evaluate the expression Add(Var(”y”), Num(1)). At first, the current value of y is read from the store and returns Choice(A, One(2), One(3)). Now, the important part is, that the addition is performed twice, incrementing both sides of the Choice construct. On the other hand, when incrementing x, the interpreter behaves differently. In this case the interpreter has to evaluate Add(Var("x"), Num(1)), but looking up x in the store returns just One(1), which is not dependent on a certain configuration. In this case, the variability-aware interpreter executes the addition only once, thus acting like a traditional interpreter and saving effort compared to the brute-force approach.

Figure 3.2 illustrates, how the variability-aware interpreter handles assignments and stores conditional values in a variability-aware store. However, to provide a more comprehensive approach, the variability-aware interpreter must support more language constructs, such as control structures. The variability-aware interpreter actually is implemented to support a While language including functions. Though it is desirable to support high-level programming languages as Java, C, C++, etc. in the future, this work presents a prototype interpreter for initial insights on viability and performance. Adapting the structures presented in this work, to high-level programming languages, is left open as a future research topic.

While executing assignments is a relatively straightforward task for the variability-aware interpreter, handling control structures requires more complicated reasoning about variability. For example, consider a *while* loop, where the number of iterations depends on the configuration. As a result, all statements in the loop body must be executed in a specific feature context, that describes in which configurations the current loop iteration is still performed. A more detailed example for *while* loops is shown below. The following paragraphs step through all language constructs, excerpt assignments, which have already been shown, and describe how the variability-aware interpreter executes them, taking variability into account during interpretation.

If statements. On interpreting *if* statements, traditional interpreters evaluate a condition and then execute either the then branch or the else branch of the statement, according to the result of the condition evaluation. Looking at code without variability, the condition always evaluates to exactly one result: *true* or *false*. When interpreting variability-enriched code, the value of the condition may depend on the configuration, which implies that possibly both branches must be visited.

The variability-aware interpreter solves this problem by firstly checking, in which configuration the condition yields *true*. The outcome of this query is subsequently used as the variability context in the execution of the then branch. If there is an else branch, it is executed with the variability context, in which the condition yields *false*. Due to that behavior, the variability-aware interpreter covers all configurations concerning *if* statements. An example for variability getting propagated by *if* statements, is shown in Figure 3.3. The value of variable c is 2 in configurations where A is selected and 3, when A is not selected. As a result, the outcome of the if statement (Lines 8-12) is also dependent on the configuration.

```
1  begin
2  //#if A
3  c = 0;
4  //#else
5  c = 1;
6  //#endif
7
8  if (c > 0) {
9    res = 3;
10 } else {
11   res = 2;
12 }
13 end
```

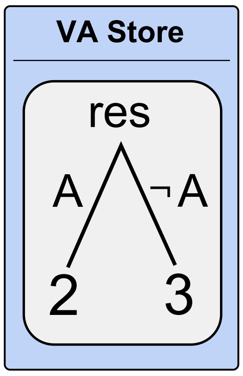


Figure 3.3: Variability propagation in *if* statements.

The variability-aware interpreter executes the then branch (Line 9) in the variability context, where the condition yields *true* and the else branch (Line 11) in the context, where it yields *false*, which is the negation of the context for the then branch. Because c is only greater than 0, when A is not selected, the context for the then branch is $\neg A$ and the context for the else branch is A. Thus, the assignments store its values only in the variability context, in which they have been executed, resulting in the illustrated store, where *res* has value 2 on feature A being selected, and value 3 otherwise.

While loops. The fact that conditions can be dependent on configurations causes the problem that *while* loops may have different numbers of iterations in variability-aware interpretation. Therefore, *while* loops are executed until there is no further possible configuration, in which the loop would perform another iteration. Additionally, the statements in the loop body are executed in the variability context, where the condition yields *true*. Considering a case, where the condition is *true* in one configuration and *false* in a different one, the variability-aware interpreter would perform the statements in the loop body only in the variability context with the condition being *true*.

Figure 3.4 shows an example for *while* loops, where the condition depends on feature A. In every iteration, the interpreter resolves the feature context, in which the condition yields *true*. For the first three iterations, $\text{i} > 0$ is *true* in every configuration. The query for the feature context in which the condition holds, therefore results in the propositional formula *true*. However, in the fourth iteration, the condition only yields *true* in variability context $\neg A$. Due to that, the incrementation of *res* (Line 9) is executed in variability context $\neg A$, which propagates the variability for variable i to *res*. After the assignment, the value of *res* has been increased by one, but only in configurations, where feature A is not selected, which is shown in the store right to the code fragment.

```
1  begin
2  //#ifdef A
3  i = 3;
4  //#else
5  i = 4;
6  //#endif
7  res = 0;
8  while(i > 0) {
9    res = res + 1;
10   i = i - 1;
11 }
12 end
```

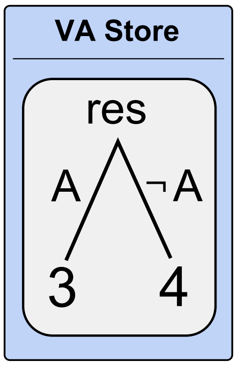


Figure 3.4: Variability propagation in *while* loops.

Block statements. On executing block statements it is important to propagate variability of a whole block to its inner statements. When blocks are executed in a specific variability context, all inner statements share this context on execution. As statements in blocks are also optional, their own variability context must be merged with the outer context. In other words, the execution of statements in a block depends on the block's variability context *and* the statements context. Because variability contexts are propositional formulas, they can be concatenated with the and operator. Just like presence con-

```
begin
x = 0;
//#ifdef A
{
  x = 1;
  //#ifdef B
  x = 2;
  //#endif
}
//#endif
end
```

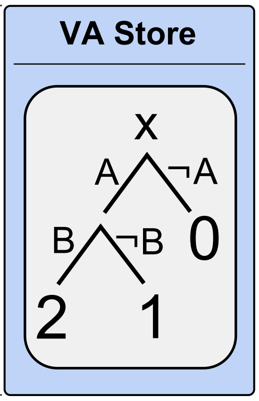


Figure 3.5: Variability propagation in blocks.

ditions, which have been described in Section 2.4.2, variability contexts are represented by the type FeatureExpr that already provides functions for concatenating propositional formulas out of the box.

In Figure 3.5, multiple assignments are executed in different variability contexts. The code fragment shows a block in feature context A (Lines 3-10) with two inner assignments. On executing the block, the variability context of the block, which is A, has to be merged with the context of its inner statements. The first inner assignment has no local variability context, which means that its context is $True$. It is therefore executed in variability context $A \wedge True$. The second assignment in the block is analogously executed in variability context $A \wedge B$. This finally results in three different values for x. As the illustrated store shows, x has value 2, when features A and B are selected, value 1, if only A is selected, and 0, when none of the two is selected.

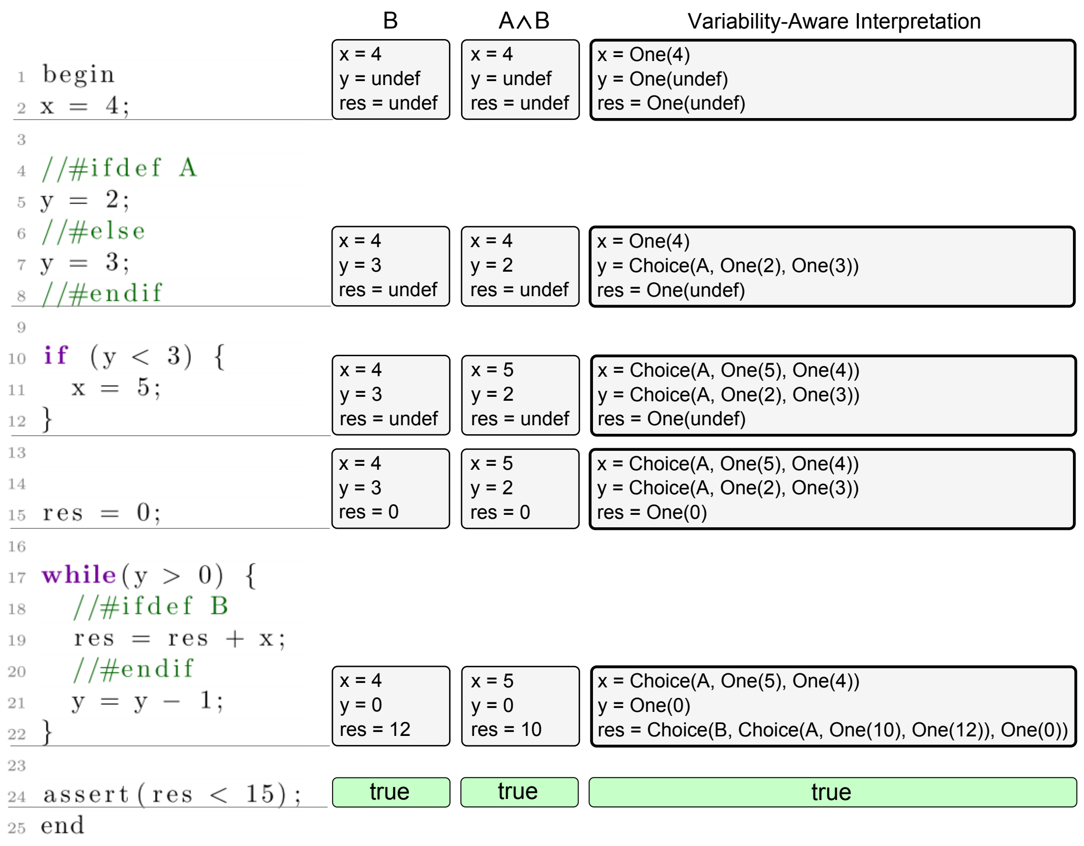
Assertions. To perform analysis on whole product lines, a construct for checking specific properties is needed. Therefore, programmers may use the *assert* statement to check if a certain condition holds.

However, assertions are statements and therefore may be executed in a certain variability context. This requires the interpreter to take the feature context, in which the assertion was executed, into account when checking a property. Figure 3.6 shows a program fragment with three assertions. Two of them are being executed in variability context A, just after an assignment to variable x (Line 5). After this assignment, x has value 5 in configurations where A is selected. Checking for $\texttt{x} == 0$ in Line 6 yields $false$, because the assertion itself is executed in variability context A. By the same reason, checking the property $\texttt{x} == 5$ (Line 7) yields $true$, when executed in the same context.

```
1 begin
2 x = 0;
3 assert(x == 0);      true
4 //#ifdef A
5 x = 5;
6 assert(x == 0);      false
7 assert(x == 5);      true
8 //#endif
9 end
```

Figure 3.6: Variability affecting assertions.

After looking at the language constructs, which are supported by the variability-aware interpreter, a more complicated example is shown, where all of the above-mentioned constructs have been used. Figure 3.7 shows the partial trace of a full program execution, allowing to see the difference between the interpretation of two exemplary configurations of the program and the variability-aware interpretation. All program variables are tracked and shown at five points of program execution. After the first assignment (Line 2) all variables are shared between configurations, but when y is assigned, the effects of variability get visible (Line 8). In case of the configurations, different values are stored for y and the variability-aware interpretation stores y as a Choice. At Line 10, the condition's value depends on the configuration, which results in the value of x also being conditional. The last snapshot of the variables illustrates the different kinds of results, which both approaches provide. While the configurations both have a unique result, the variability-aware interpretation yields the result for all configurations at once. The assertion in Line 24 checks the property $\texttt{res} < 15$, which is $true$ in all cases. While this outcome is obvious in case of the configurations, because their result is unique, the variability-aware interpretation checks all values for res to be less than 15. In this case, res can have values 0, 10 and 12. Therefore, the assertion evaluates to $true$.



```
begin
x = 4;

//#ifdef A
y = 2;
//#else
y = 3;
//#endif

if (y < 3) {
  x = 5;
}

res = 0;

while(y > 0) {
  //#ifdef B
  res = res + x;
  //#endif
  y = y - 1;
}

assert(res < 15);
end
```

B	A∧B	Variability-Aware Interpretation

x = 4
y = undef
res = undef
	x = 4
	y = undef
	res = undef
		x = One(4)
		y = One(undef)
		res = One(undef)

x = 4
y = 3
res = undef
	x = 4
	y = 2
	res = undef
		x = One(4)
		y = Choice(A, One(2), One(3))
		res = One(undef)

x = 4
y = 3
res = undef
	x = 5
	y = 2
	res = undef
		x = Choice(A, One(5), One(4))
		y = Choice(A, One(2), One(3))
		res = One(undef)

x = 4
y = 3
res = 0
	x = 5
	y = 2
	res = 0
		x = Choice(A, One(5), One(4))
		y = Choice(A, One(2), One(3))
		res = One(0)

x = 4
y = 0
res = 12
	x = 5
	y = 0
	res = 10
		x = Choice(A, One(5), One(4))
		y = One(0)
		res = Choice(B, Choice(A, One(10), One(12)), One(0))

true	true	true

Figure 3.7: Partial trace of program execution for two specific variants and in variability-aware fashion.

It already has been said, that variability-aware interpretation is expected to be faster than checking all possible configurations individually, because common code segments are executed only once. Two special cases of this effect that emphasize the profitability of variability-aware interpretation are shown below.

Early joining. An effect that occurs, when previously configuration dependent values are joined again, is called early joining. To join values in this case means, that the number of configurations, in which they are equal, increases.

The benefit of joining as early as possible is, that less subsequent calculations must be performed to provide a result for all configurations. An example of early joining is shown in Listing 3.1. In this code fragment, two functions are declared: max(a,b) returns the maximum value of a and b, while expensive(n) is a placeholder representing any sort of CPU-intensive calculation. When calling the max function (Line 10), the value of x is conditional. However, the max function returns the same value for every configuration and therefore x is shared between configurations again, after this assignment. Due to that fact, the expensive calculation (Line 11) has to be applied only once, increasing the variability-aware interpretation speedup.

```
1  begin
2  def expensive(n) { ... }
3  def max(a,b) { ... }
4
5  x = 1;
6  //#ifdef A
7  x = 2;
8  //#endif
9
10 x = max(x, 42);
11 res = expensive(x);
12 end
```

Listing 3.1: Code example for early joining.

Late splitting. When expensive calculations differ only slightly between configurations, intermediate results may have to be calculated only once. The goal is to keep values shared between configurations as long as possible and only split them for the configuration-dependent part of the calculation. This effect is called late splitting and in contrast to early joining, which is implemented as a special feature, it occurs by construction. Late splitting is illustrated in Listing 3.2.

The code fragment shows the calculation for the sum of the first n natural numbers. The calculation mainly consists of a *while* loop, summarizing all numbers between 1 and x. In every iteration the intermediate result is stored in variable sum. Previously, x has been assigned (Lines 2-5) and affects the number of iterations, the *while* loop will perform (Line 9). Above, it has been mentioned, that the variability-aware interpreter executes *while* loop bodies in the context, where the while condition yields true. In Listing 3.2, this context is always $True$, except for the last iteration. As a consequence, the intermediate result, which is stored in sum, is updated once in nearly all iterations. Only the last iteration is executed in variability context A, thus updating sum conditionally. Also additional effort for handling variability is only required in this last iteration. In comparison to the brute-force approach, which would execute the whole *while* loop in every possible product, late splitting makes up a significant performance difference.

```
1  begin
2  x = 100;
3  //#ifdef A
4  x = 101;
5  //#endif
6
7  sum = 0;
8  i = 1;
9  while(i <= x) {
10   sum = sum + i;
11   i = i + 1;
12 }
13 end
```

Listing 3.2: Code example for late splitting.

3.3 Implementation

In this section the concept of the variability-aware interpreter is picked up again and its implementation is shown. Especially, implementations of the AST classes, the parser, the execution of statements, the evaluation of expressions and the variability-aware store are shown. The variability-aware interpreter is written in Scala to enable compatibility with the TypeChef libraries, which are also written in Scala.

AST

In Section 2.4.1 it has been described, how the source code of programs is represented using ASTs. Afterwards, Section 2.4.2 showed, how variability is woven into ASTs when the code they represent is enriched with variability. In the following, it is illustrated how the AST nodes are implemented as constructs of Scala.

```
abstract class Stmt
case class Assign(n: String, e: Conditional[Expr]) extends Stmt
case class If(c: Conditional[Expr], thn: Block, els: Block) extends Stmt
case class While(c: Conditional[Expr], b: Block) extends Stmt
case class Block(s: List[Opt[Stmt]]) extends Stmt
case class FuncDec(n: String, a: List[Opt[String]], b: Block) extends Stmt

abstract class Expr
case class Num(i: Int) extends Expr
case class Var(name: String) extends Expr
case class Call(n: String, a: List[Opt[Expr]]) extends Expr
case class Add(e1: Expr, e2: Expr) extends Expr
case class Sub(e1: Expr, e2: Expr) extends Expr
...
```

Listing 3.3: Language constructs of a variability-enriched While language in Scala.

As shown in Listing 3.3, the While language constructs are divided in statements and expressions. Statements include Assignments, which have a field for the variable name and an expression, which will be evaluated and stored as value for the corresponding variable. If- and While statements have a condition, which will be evaluated before their bodies are executed. While loops have one block statement, that includes the statements which will be repeatedly executed. *If* statements have one block for their then branch and another block for the else branch. Function declarations also count as statements and include the function name, a list of function parameters and the function body as a block statement. Expressions can be numbers (Num), the value of a variable (Var) or function calls (Call). Calls have the name of the called function as field, as well as a list of parameter expressions. Expressions can also be operators (Add/Sub/...), which include another left and right Expression. The abstract syntax of the While language is extended with Variability by using Opt and Conditional constructs, which have been described in Section 2.4.2. This provides the ability to express, for example, function declarations with configuration dependent arguments.

Parser

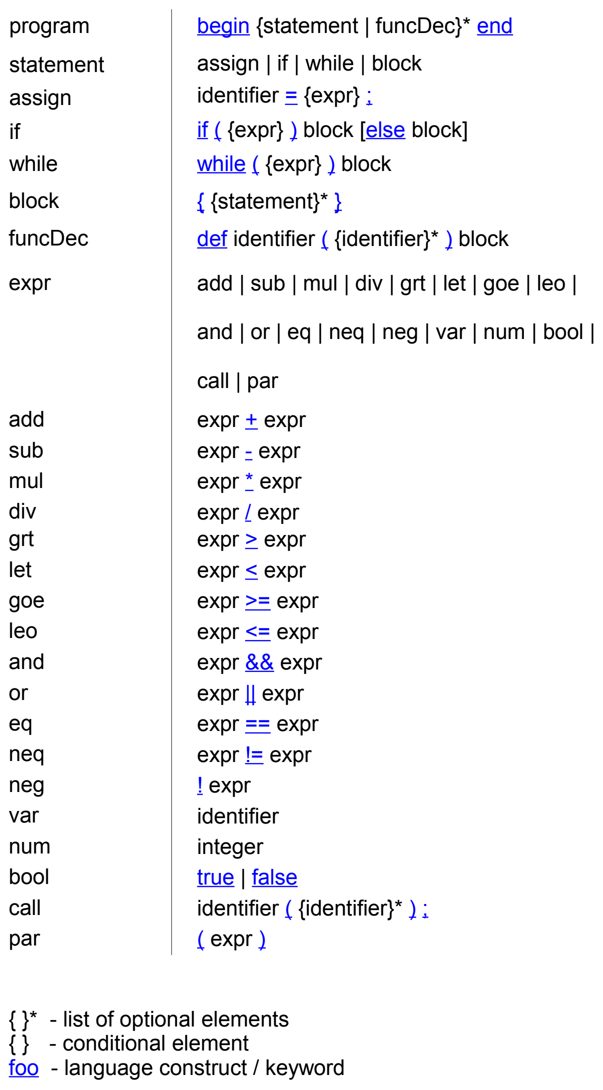
To transfer source code into its corresponding representation of the above-mentioned classes, the code is processed by a parser. The concept of the parser has already been described in Section 2.5. In this section, a small example is shown, that illustrates how a textual code representation is finally transferred to objects of the abstract syntax classes.

```
1 def whileStmt = "while" ~ "(" ~ (expr !) ~ ")" ~ block ^^ {
2   case _~_~c~_~b => While(c, b)
3 }
4 def blockStmt = "{" ~ (stmt *) ~ "}" ^^ {
5   case _~stmtlst~_ => Block(stmtlst)
6 }
```

Listing 3.4: Parsing while- and block statements with the TypeChef Parser Framework

Listing 3.4 is an excerpt from the parser implemented for this work's purposes. It shows how parsing while- and block statements is realized using auxiliary operators of the TypeChef Parser Framework [Kästner et al., 2011b]. How exactly these operators work, is not part of this work and can be looked up in the referenced literature. Concerning this work, TypeChef's parser functions and operators have been used out of the box, as already stated in Section 2.5.

A While statement (Lines 1-3) consists of the keyword "while", a condition in braces that is parsed using expr and a block. A pair of curly braces that wraps a list of optional statements are the constructs for a Block (Lines 4-6). The different elements, that belong to these constructs are concatenated using a sequence operator (~). The condition in a *while* loop is conditional, which means that it can differ between configurations, which is achieved using the ! operator after expr. An example for the usage of optional lists ($*$) is given in the definition of a block statement, as it contains an arbitrary number of optional statements. Both constructs are mapped to their corresponding AST node representation on parsing. Therefore, the ^^ operator is used to create a While object that contains the parsing result of its condition and its block, and a Block object that contains the parsing result of its list of statements.



program	begin {statement | funcDec}* end
statement	assign | if | while | block
assign	identifier = {expr} ;
if	if ({expr}) block [else block]
while	while ({expr}) block
block	{ {statement}* }
funcDec	def identifier ({identifier}*) block
expr	add | sub | mul | div | grt | let | goe | leo | and | or | eq | neq | neg | var | num | bool | call | par
add	expr + expr
sub	expr - expr
mul	expr * expr
div	expr / expr
grt	expr > expr
let	expr < expr
goe	expr >= expr
leo	expr <= expr
and	expr && expr
or	expr || expr
eq	expr == expr
neq	expr != expr
neg	! expr
var	identifier
num	integer
bool	true | false
call	identifier ({identifier}*) ;
par	(expr)

{ }* - list of optional elements
{ } - conditional element
foo - language construct / keyword

Figure 3.8: WHILE language syntax.

Finally, Figure 3.8 describes the syntax of the variability-enriched While language that is accepted by the parser of this work. A program contains a list of optional statements and function declarations. It starts with "begin" and is closed with "end". All these constructs are implemented using the TypeChef Parser Framework and its helper functions, analogously to the example presented in Listing 3.4.

Statements

After parsing the source code of a program, it is stored as a list of optional statements. Running the program means, that the interpreter successively processes all statements and accordingly updates its store. Listing 3.5 shows the implementation of statement execution in the variability-aware interpreter. First of all, the interpreter does not execute any statement, if the variability context is a contradiction (Line 2). The handling for the remaining statement types is explained in the following.

Assignments. On interpreting an assignment (Lines 4-6), at first, the right side of the assignment is evaluated to a conditional value. Afterwards this value is saved in the store. The value is saved as a Choice to express, that the assigned variable receives its value only in the variability context, the assignment has been called with. Finally, if this would yield an unnecessarily complicated structure, it is simplified with the corresponding function (Line 6). An example for this process would be simplifying Choice(True, One(1), One(2)) to One(1).

Blocks. When interpreting blocks, all statements included in a block are executed in the variability context of the block itself *and* their own context. To achieve this, the interpreter is recursively called with variability context $fe \wedge vctx$ (Line 8).

While loops. The concept of how *while* loops have to be handled concerning variability, has been described in Section 3.2. In short, every iteration of the loop needs to query the variability context, in which the *while* condition yields true. Looking at the code, this behavior is received by calling *whenTrue* on the result of evaluating the condition in every iteration (Line 13). The result of *whenTrue* (Lines 35-40) is a feature expression, that expresses all configurations, in which the evaluated condition is *true*. The loop is exited, when there is no configuration left, in which another iteration could be performed.

If statements. For interpreting if statements, the variability context in which the if condition is *true*, is also queried (Line 19). When this context is satisfiable, the *then* branch is executed with it (Line 20). If there are statements in the else branch, these are executed in the negation of the queried context, which is the context, where the condition yields $false$ (Line 21).

Assertions. Interpreting assertions, an exception is thrown, when the variability context of when the assertion condition yields *true*, is not equivalent to the context, the assertion has been called in (Line 25).

Function declarations. When the interpreted statement is a FuncDec (Lines 29-31), it is stored in the function store of the execution. The behavior is the same, as with assignments to variables, which has been explained above.

```scala
def execute(s: Stmt, vctx: FeatureExpr, sto: VAStore, fSto: VAFuncStore) {
  if (vctx.isContradiction()) return
  s match {
    case Assign(name, expr) =>
      sto.put(name, Choice(vctx, eval(expr, sto, fSto), sto.get(name))
              .simplify)
    case Block(stmts) => {
      for (Opt(fe, stm) <- stmts) execute(stm, fe and vctx, sto, fSto)
    }
    case While(c, block) => {
      var isSat = true
      while (isSat) {
        val x = whenTrue(eval(c, sto, fSto))
        isSat = (vctx and x).isSatisfiable
        if (isSat) execute(block, iterctx, sto, fSto)
      }
    }
    case If(c, s1, s2) => {
      val x = whenTrue(eval(c, sto, fSto))
      if (x.isSatisfiable) execute(s1, vctx and x, sto, fSto)
      if (!(s2.stmts.isEmpty)) execute(s2, vctx andNot x, sto, fSto)
    }
    case Assert(c) => {
      val x = whenTrue(eval(One(c), sto, fSto))
      if (!(x.equivalentTo(vctx))) {
        throw new AssertionError("violation of " + printNode(cnd))
      }
    }
    case FuncDec(name, args, body) =>
      fSto.put(name, Choice(vctx, One(FDef(args, body)),fSto.get(name))
               .simplify)
  }
}

def whenTrue(cv: Conditional[Value]): FeatureExpr =
  cv match {
    case One(ErrorValue(_)) => False
    case One(v) => if (v.getBool) True else False
    case Choice(f, a, b) => (whenTrue(a) and f) or (whenTrue(b) andNot f)
  }
```

Listing 3.5: Method for processing statements in the variability-aware interpreter.

```scala
trait Value {
  def getInt(): Int
  def getBool(): Boolean
}

case class IVal(i: Int) extends Value {
  def getInt() = i
  def getBool() = throw new IllegalCallException()
}

case class UndefVal extends Value {
  def getInt() = throw new IllegalCallException()
  def getBool() = throw new IllegalCallException()
}
```

Listing 3.6: Value representation in the variability-aware interpreter.

Expressions

On executing statements, the variability-aware interpreter relies on evaluating expressions. The outcome of evaluating an expression can be a conditional boolean or integer value. In the variability-aware interpreter, values are implemented as subtypes of the trait Value. As shown in Listing 3.6, these subtypes can get queried for integer and boolean values. However, the example of IVal shows, that calling the false getter results in an exception (Line 8). There is also a type UndefVal, that represents an undefined value. When undefined values occur in a calculation, the error propagates. This behavior is preferable to throwing an exception, because continuing the evaluation provides the possibility to detect error-containing configurations afterwards and separate them from others, that provided a correct result.

The evaluation of conditional expressions is shown in Listing 3.7. To map over a conditional structure, the method *mapr* from the TypeChef library is used, which provides the possibility to process every alternative expression in its separate variability context. An example of this behavior is shown in Figure 3.9, where *mapr* is called with the functional parameter $store.get(\_)$ on a conditional variable name (left). This results in all alternative variable names being looked up in the store separately and preserving the variability context C of the queried variable y (right).

Listing 3.7 also shows the handling of the different expression types. When the expression is a plain number (Num), the corresponding result is a single IVal (Line 3). When the type of the expression is a variable name (Var), the value of this variable is looked up in the store (Line 4). For evaluating composite expressions (Lines 5-6), an auxiliary function $calcVal$ is used, which first evaluates the inner expressions, propagates possible error values and finally executes the given operation on the outcome of its inner evaluations. In Listing 3.7, only Add and Sub are shown, because the handling for other composite expressions is the same.

Finally, the processing of function calls is shown. In this case another call for *mapr* is needed, because looking up a function in the function store, yields a conditional function. This is desirable, because one function can have alternative implementations, according to a certain configuration. However, when there are alternative implementations, the call must be processed for each implementation of the function. It may also happen, that a function is not defined at all in a certain configuration, which results in an UndefVal being returned (Line 11). When the function is actually defined, a new local store for the function execution is created (Line 16), that includes the variables from the function definition mapped to the evaluated expressions from the function call. Because optional function parameters are supported for the definition, as well as the call, these variability contexts have to be merged before (Lines 19-22). Afterwards, the function body is executed using the new local store (Line 23). Return values are realized as assignments to a variable *res* inside of the function body. If there is no such variable, the call returns a UndefVal, informing that the returned value is of type *void*.

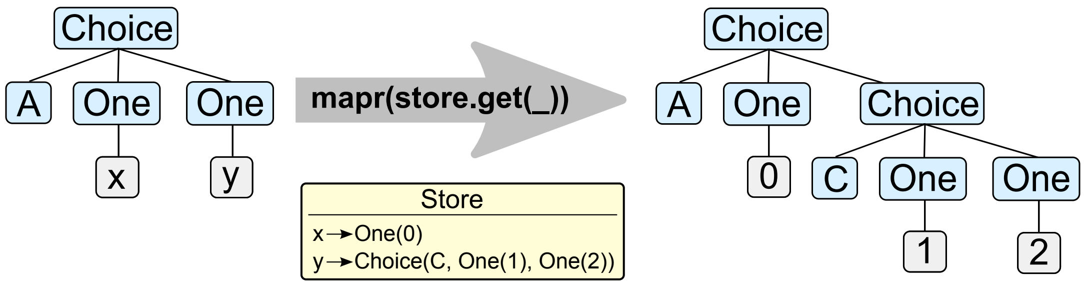


Figure 3.9: Exemplary mapping over conditional structures.

```scala
def eval(exp: Conditional[Expr], sto: VAStore, fSto: VAFuncStore) = {
  exp.mapr(_ match {
    case Num(n) => One(IVal(n))
    case Var(name) => sto.get(name)
    case Add(e1,e2) => calcVal(e1,e2,(a,b) => IVal(a.getInt+b.getInt))
    case Sub(e1,e2) => calcVal(e1,e2,(a,b) => IVal(a.getInt-b.getInt))
    ...

    case Call(cname, cargs) => {
      fSto.get(cname).mapr(_ match {
        case FErr(msg) => One(UndefVal(msg))
        case FDef(fargs, fbody) => {
          if (cargs.size != fargs.size)
            One(UndefVal("illegal arguments size"))
          else {
            val locStore = new VAStore()
            for (i <- 0 until cargs.size)
                locStore.put(fargs(i).entry,
                  Choice(args(i).feature and fargs(i).feature,
                         eval(One(args(i).entry), sto, fSto),
                         One(UndefVal("undef func arg"))
                         ).simplify)
            execute(fbody, True, locStore, fSto)
            if (!locStore.contains("res"))
              One(UndefVal("'" + name + "' of type void"))
            else
              locStore.get("res")
          }
        }
      })
    }
  })
}
```

Listing 3.7: Method for evaluating expressions in the variability-aware interpreter.

Store

To map values to variables, the interpreter interacts with a variability-aware store. The implementation of the store is shown in Listing 3.8. Basically, the store consists of a simple Map[String, Conditional[Value]], but behaves differently, when a variable is queried, which is not yet in the store. In this case, a single UndefVal is returned (Line 6). Also, it is possible, to get the value of a variable in a specific variability context. This is implemented using the TypeChef helper function *findSubtree*, which looks for values that fit to variability context *vctx* in a conditional construct (Line 12).

```
1  class VAStore {
2    val vars = Map.empty[String, Conditional[Value]]
3
4    def put(key: String, value: Conditional[Value]) = vars.put(key, value)
5
6    def get(key: String) = vars.getOrElse(key, One(UndefVal(s)))
7
8    def getByContext(key: String, vctx: FeatureExpr) =
9      if (!vars.contains(key))
10       One(UndefVal(s))
11     else
12       findSubtree(vctx, vars.get(key).get)
13 }
```

Listing 3.8: Implementation of the variability-aware store.

Plain interpreter

Next to the variability-aware interpreter, a plain interpreter has been implemented, that is able to interpret programs without variability contexts. Reasons for implementing a second interpreter, are the possibility to perform comparisons between both implementations (Chapter 4) and the fact, that the existing variability-enriched AST classes can be reused with an adjusted handling. When the plain interpreter is called, it completely ignores variability, avoiding all checks for variability contexts and thus providing a traditional interpreter. However, the fact that it reuses the existing AST structures requires some modifications, which are exemplarily shown for assignments and *while* loops in Listing 3.9. For example, on interpreting assignments, the interpreter assumes unambiguous expressions (Line 3) and uses a plain store, which maps variable names to values instead of conditional values (Map[String, Value]). Also, conditions in *while* loops and *if* statements are unambiguous. For that reason, the evaluated condition implies directly, whether the loop is continued or not (Lines 6-12). On the contrary, the variability-aware interpreter would need to check in which variability context further iterations are executed, at this point.

```
1  def execute(s: Stmt, sto: PlainStore, fSto: PlainFuncStore) {
2    s match {
3      case Assign(name, One(expr)) => sto.put(name, eval(expr, sto, fSto))
4      case While(One(c), s) => {
5        var cnd = true
6        while (cnd) {
7          cnd = eval(c, sto, fSto) match {
8            case ErrorValue(_) => false
9            case v => v.getBool
10         }
11         if (cnd) execute(s, sto, fSto)
12       }
13     }
14     ...
15   }
16 }
```

Listing 3.9: Execution of statements in the plain interpreter.

Chapter 4

Evaluation

In the following chapter the variability-aware interpreter is evaluated. The evaluation focuses on providing a comparison between the traditional brute-force approach and the interpreter introduced in this work. Therefore, three benchmarks are performed. Two of them focus on benchmarking specific cases, the third one is applied to a set of 100 randomly generated test product lines. In every benchmark, the speedup of the variability-aware interpreter in relation to the complexity of the tested product-line is shown. Initially, Section 4.1 describes the evaluation method used in this work. Afterwards, the strategy used for comparing both approaches is explained in Section 4.2. Section 4.3 describes which subjects are used for the evaluation and why, before the actual results are shown in Section 4.5. Finally, Section 4.6 discusses the results und points out possible threats to validity.

4.1 Method Selection

When performing empirical evaluations, a variety of possibilities to measure the quality of an approach exist. The problem addressed in this work, is to provide an approach for testing whole product-lines, which yields an exhaustive result, but scales better in performance than brute-force analysis, for high numbers of features. Therefore, the size under investigation is the runtime of both approaches, concerning interpreted product lines. In search of an adequate evaluation method, *performance benchmarks* have been chosen, because runtimes on a real system are required. Performance benchmarks as evaluation method, focus on actually executing programs on such a system. This method fits for evaluating this work, because interpreters are programs, that interpret and execute source code, and which can be executed on a test system to measure the time elapsed for completing the interpretation of a certain code fragment.

4.2 Comparing Runtimes

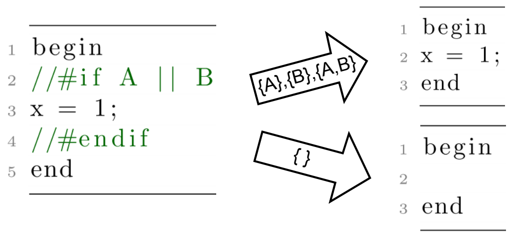
To compare the runtime of the variability-aware interpreter and the brute-force analysis, a specific comparison method is necessary. Gathering the correct runtimes requires more than just invoking two interpreters on a certain code fragment, because the brute-force approach must be applied on all possible variants of a product line under test. Applying the brute-force approach in this case, means invoking the plain interpreter, which has been already described in Section 3.3. The plain interpreter has been designed especially for evaluation purposes, because the impact of variability calculations would distort the benchmark results, when the variability-aware interpreter was also used to execute the variants in the brute-force approach, though these programs do not contain variability anymore. However, when using the plain interpreter, all statements and expressions are assumed unambiguous by construction. The interpreter does not take a variability context, nor does it own any functions for processing variability. It handles statements and expressions like a traditional interpreter, which is desired when comparing variability-aware interpretation with the brute-force approach. The strategy we used for receiving the correct runtime values for both approaches can be described in the following major steps:

1. Generate possible variants.
   In the first step, all possible variants from a certain product line under test have to be created. This is done by looking up all distinct features, that have been used in the corresponding product line and creating a set of all possible feature selections, that can be made. Afterwards, a variant of the product line according to every of the above-mentioned feature selections is generated. After removing duplicates, this process yields all possible variants, that can be derived from the product line under test.

2. Execute the interpreters.
   After generating all possible products, the actual runtime values are collected. This work measures wall-clock time of the pure program execution, which means without parsing. The exact method is shown in Listing 4.1. As it shows, system time is measured before and after program execution and the difference is taken as the runtime of program execution. This way of testing is called differential testing. In case of the variability-aware interpreter, only one execution of the program has to be measured. Concerning the plain interpreter, the runtime for every variant execution is measured and summarized.

Figure 4.1 shows a minimal example for generating all possible program variants. Though four possible configurations exist, when combining the optional features A and B, all configurations where at least one feature is selected yield the same variant (top right). Only when no feature is selected, the resulting program is empty (bottom right), which results in a total of 2 possible variants.

At the end of the second step, the results for both approaches are finally collected. This way of comparing both approaches also conforms to the comparison of variability-aware analysis and brute-force analysis (Figure 2.3), which has been described in Section 2.4.



```
begin
//#if A || B
x = 1;
//#endif
end
```

```
begin
x = 1;
end
```

```
begin

end
```

Figure 4.1: Generating all possible program variants.

```
start = System.nanoTime()
program.run()
stop = System.nanoTime()
total = (stop − start)
```

Listing 4.1: Measuring wall-clock time of interpreter execution.

In the following Sections, the expression ”speedup value” is frequently used. It refers to the runtime of the brute-force approach divided by the runtime of the variability-aware interpretation, concerning a certain product line. The speedup value describes the factor of which the variability-aware approach is faster than the brute-force approach.

4.3 Subject Selection

Empirical evaluations always need subjects on which the evaluation can be performed. In case of qualitative analyses, this can be a group of probands. This work, however, uses performance benchmarks as evaluation method and therefore needs some kind of data, on which the benchmarks can be performed. The kind of data addressed by this work are product lines. Therefore adequate product lines for conducting performance benchmarks have to be selected. Which product lines serve as subjects for the evaluation is explained in the following.

4.3.1 Benchmarking specific cases

In Section 3.2, it has already been explained why keeping variability as local as possible and thus sharing structures between configurations are benefits of the variability-aware approach in comparison to the brute-force approach. Early joining and late splitting have been introduced as positive effects that occur, when code fragments adapt a specific pattern. In the evaluation of this work, a benchmark is conducted for both cases, to provide a speedup value for each effect in isolation. Also, these cases have been selected, because the variability-aware interpreter can unfold its full potential, on appearance of the above-mentioned effects, which should be emphasized with some actual speedup values.

```
begin
def fibo(a) {
    if (a == 1 || a == 2) {
      res = 1;
    } else {
      res = fibo(a-1) +
         fibo(a-2);
    }
}

def max(a,b) {
   if (a > b) {
     res = a;
   } else {
     res = b;
   }
}

x = 1;
//#ifdef A
x = 2;
//#endif
//#ifdef B
x = 3;
//#endif
...

x = max(x, 25);
res = fibo(x);
end
```

Listing 4.2: Code fragment used for measuring the effect of early joining.

Early joining

Early joining exploits the fact that values are joined, before expensive calculations are executed on them. In this benchmark, a function for the nth fibonacci number represents the expensive calculation. The value for variable x is joined when the function max(a, b) is executed, because it was less than 25 in all configurations before. The benchmark is conducted with multiple starting values of x. Initially, the code is interpreted without any variability. All further executions add another assignment below 25 to x to the code, which is executed in a specific variability context, thus increasing the expected benefit from early joining in favor of variability-aware interpretation. Further assignments increase the assigned value by one and use the subsequent alphabetical character as variability context. Listing 4.2 shows an example of the benchmark, where x is currently dependent on two features. An increasing speedup is expected, because the expensive calculation (Line 28) is executed once in the variability-aware interpreter, while it has to be executed multiple times, according to the number of possible products, in the brute-force approach.

Late splitting

When calculations use a single variability context as long as possible and only split, when variability actually affects the calculation result, this effect is called late splitting. In this benchmark the impact of late splitting is measured based on the code fragment in Listing 4.3. The code fragment used for benchmarking the impact of late splitting is very close to the code, that has been used for explaining the effect in Section 3.2. Still, the calculation for the sum of the first n numbers is used. However, in this benchmark the number of assignments to x is increased step by step. Starting off with no specific variability context, for further executions another assignment to x in a alphabetically increasing variability context is added. In this way, the impact of late splitting is expected to increase, because in variability-aware interpretation, the calculation is executed in a single variability context until 100 and only then splits for the different variability contexts. On the contrary, the brute-force approach executes the whole calculation for every possible product. Because an increasing number of different variability contexts increases the calculation effort significantly less for variability-aware interpretation compared to the brute-force approach, the speedup of the variability-aware interpreter is also expected to increase.

```
begin
x = 100;
//#ifdef A
x = 101;
//#endif
//#ifdef B
x = 102;
//#endif
...

sum = 0;
i = 1;
while(i < x) {
  sum = sum + i;
   i = i + 1;
}
end
```

Listing 4.3: Code fragment used for measuring the effect of late splitting.

4.3.2 Benchmarking generated product lines

The third benchmark performed in this work, includes testing 100 random generated test product lines. In contrast to the previously described benchmarks, which show the variability-aware interpreter's potential in favorable cases, this benchmark aims at providing a comparison for a variety of different code fragments. To provide a realistic result, the set of test subjects contains many structurally different product lines. The following list provides a more detailed insight of which structural attributes are varied within this set.

- **Number of statements.** The size of the test product lines is continually increased by extending the number of statements, which are generated.

- **Type of statements.** Not only the number of statements changes over the set of subjects, but also the type of statements that is used. Possible types of statements are assignments, if statements and while loops.
- **Number of features.** In the set of test product lines, statements are randomly annotated with variability contexts. For this purpose, a subset of the features *A*-*F* is used. Product lines with greater size use more features respectively.
- **Structure of feature expressions.** Next to varying the number of feature expressions, the structure of them is also varied. Structural variations include how many distinct features are used in a complete feature expression and how these distinct features are concatenated. For example, a feature expression can just be one distinct feature (X) or the conjunction of many distinct features by an *and* operation ($X \wedge Y \wedge$ ...).
- **Usage of functions.** Functions have also been incorporated in the set of test product lines. The number of declared and used functions for a certain product line is determined by its size, which means the number of statements. In other words, greater product lines use more functions. Though function definitions, as well as function calls are randomly generated with a certain variability context, the actual function body is taken from a static set of predefined functions, e.g. *sum*, *sub*, *min*, *max*, etc.

The generator for these product lines is written with ScalaCheck [Nilsson], a library for automated testing and test case generation. On generating product lines, ScalaCheck automatically covers a variety of special cases, such as using one of the above-mentioned constructs excessively (e.g. many while loops), or on the other hand, using it not at all (e.g. an empty product line). Therefore, the comparison of both approaches over the whole set of generated product lines, is expected to yield a representative result for the speedup of variability-aware interpretation. The amount of 100 random generated product lines, has been borrowed from the standard settings of the ScalaCheck test case generator.

4.4 Setup

The following section briefly describes the setup used for benchmark execution. All benchmarks in this work have been performed on the same workstation. The hardware and virtual machine settings of the workstation are shown in Figure 4.2. To minimize the impact of confounding factors, all unnecessary background processes have been closed prior to benchmark execution. Also, the garbage collector has been launched manually before each benchmark run and one warmup run was executed before the actual measurement. Additionally, all benchmarks were performed three times, and the results shown in the next section, represent mean values of these three runs.

```
CPU : Intel Core i7 3770K (4 x 3,5GHz)
RAM : 16GB DDR3-1600
LANG: Scala 2.9.2
VM  : Java HotSpot(TM) Client VM (1.6.0_32)
HEAP: 512MB (default) / 1024MB (max)
```

Figure 4.2: Hardware setup of the benchmark system.

4.5 Results

4.5.1 Single cases

Early joining

In the first benchmark, the impact of early joining on variability-aware interpretation speedup was the target of the evaluation. The intuition was, that an increasing number of features would also increase the variability-aware interpretation speedup, which led to the following hypotheses:

Null hypothesis (H_0): In case of early joining, the number of used features, has no relation to the speedup of the variability-aware interpreter.

Alternative hypothesis (H_1): For early joining, a relation between the number of features and the speedup of the variability-aware interpreter exists.

Figure 4.3 shows the results for the early joining benchmark. The variability-aware interpreter performs slower than the brute-force approach when the joined variable is shared between configurations or dependent on only one feature. However, starting at the use of two variability contexts, the variability-aware interpreter is faster and also increases its speedup for the further executions of this benchmark. When the joined variable was dependent on 6 features before joining, this benchmark yields a speedup of over 20 times for the variability-aware interpreter. The absolute values are around 850ms for all runs of the variability-aware interpreter and range from 280ms to 17700ms for the runs of the brute-force approach.

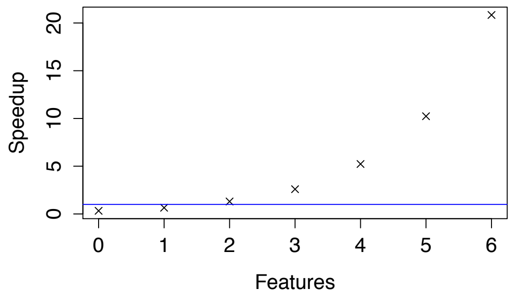


Figure 4.3: Results for early joining benchmark.

Late splitting

The second benchmark addressed the impact of late splitting on the speedup of the variability-aware approach. Again, the intuition was, that an increasing number of features affects the variability-aware interpretation speedup positively, because calculations are mostly performed in a single variability context.

Null hypothesis (H_0): When late splitting occurs, the number of used features, has no relation to the speedup of the variability-aware interpreter.

Alternative hypothesis (H_1): On occurrence of late splitting, a relation between the number of features and the speedup of the variability-aware interpreter exists.

The results for the late splitting benchmark are shown in Figure 4.4. Compared to the results of the previous benchmark, the variability-aware interpreter crosses the mark where it performs faster than the brute-force approach, at a later point. With a growing number of features, however, the speedup also increases, just like in the previous benchmark. The variability-aware interpreter reaches a speedup of over 20 times, when the variable that gets split after 100 loop iterations, is dependent on 10 different features. In this benchmark, the absolute values range from 54ms to 433ms in case of the variability-aware interpreter and from 14ms to 10100ms in case of the brute-force approach.

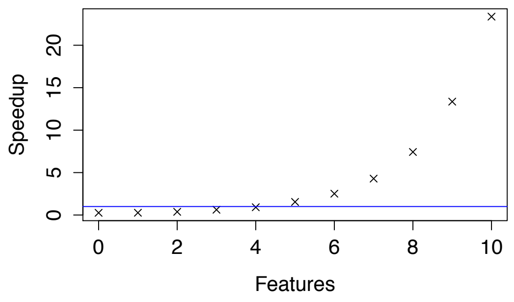


Figure 4.4: Results for late splitting benchmark.

4.5.2 Generated product lines

Finally, both interpreters were compared over the set of 100 generated product lines. In this third benchmark, the evaluation provides a more comprehensive result because the product lines under test differ in a variety of structural attributes. Though there may be some critical cases, where the variability-aware interpreter performs poorly, the expectation was, that it outperforms the brute-force approach regarding the whole set of product lines. Therefore we raised the following hypotheses:

Null hypothesis (H_0): Comparing the runtimes of all 100 generated product lines under test, the variability-aware interpreter does not perform better than the brute-force approach.

Alternative hypothesis (H_1): The variability-aware interpreter outperforms the brute-force approach, when taking all collected runtime values into account.

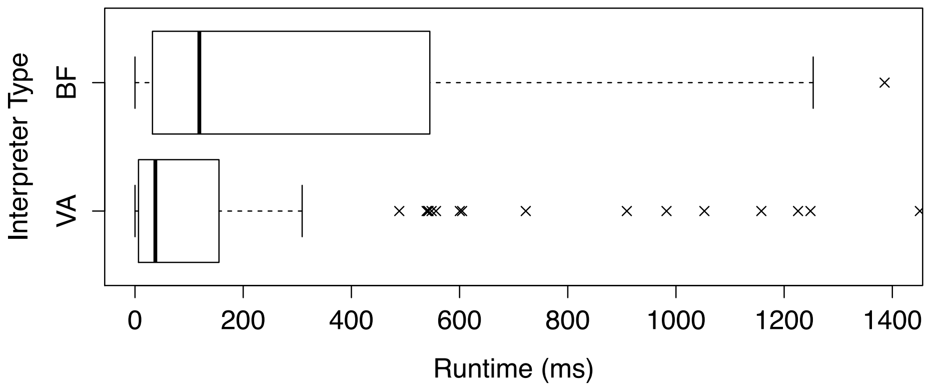


Figure 4.5: Boxplot of runtimes for generated product lines benchmark. Some outliers omitted.

A comparison of both approaches concerning plain runtimes is shown by the boxplot in Figure 4.5. The boxes on the left include fifty percent of all collected runtimes, while their inner vertical line shows the median for each approach's runtimes. Some outliers for the brute-force approach have been omitted. The boxplot shows, that the majority of collected runtime values for the variability-aware interpretation is between slightly over 0ms and 350ms. On the contrary, the majority of runtimes for the brute-force approach is spread over a greater interval, until 1300ms. The observed medians are at about 40ms for the variability-aware interpreter and about 120ms for the brute-force approach.

Figure 4.6 shows the speedup value of the variability-aware interpreter dependent on the number of variants. In the previous benchmarks, the number of features directly indicated the number of variants, because there was exactly one statement in the variability context of each feature, resulting in $2^{\#features}$ variants. In this benchmark, the subjects under test are generated randomly and it is possible, that some of the product lines yield equal variants for different feature selections, which lowers the number of possible variants, though the number of features might be the same. Therefore the speedup is shown in relation to the number of variants, in this benchmark. As the generated product lines used at most 6 distinct features, the number of variants caps at $2^6 = 64$. The blue line in the plot marks the point, at which the variability-aware interpreter outperforms the brute-force analysis. While the brute-force approach performs better for very small product lines ($\leq$ 8 variants), the variability-aware interpreter is faster in 76 out of 83 cases with more than 8 variants, receiving speedup factors of up to 18.

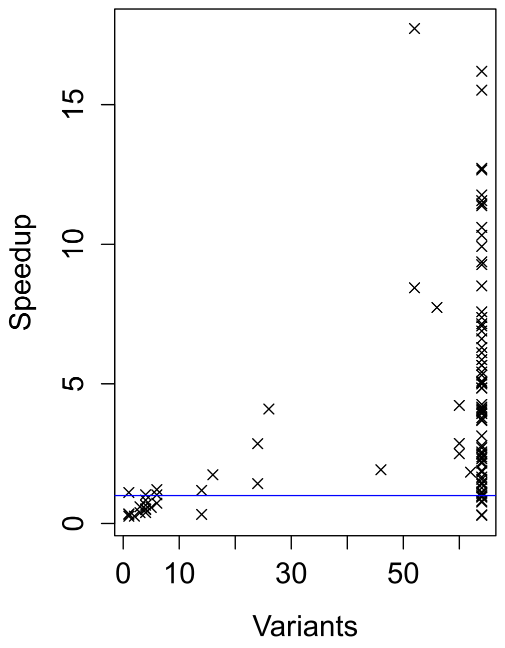


Figure 4.6: Variability-aware interpretation speedup in relation to the number of variants for generated product lines benchmark.

4.6 Discussion

In the evaluation, three benchmarks have been conducted to compare the variability-aware interpreter with the brute-force approach. The first two benchmarks addressed cases, where an advantage in favor of the variability-aware approach was expected beforehand. The null hypotheses were rejected, because an increasing number of features indeed affected the variability-aware interpretation speedup positively. As a consequence, the alternative hypotheses are valid. In the third benchmark, a set of 100 generated product lines was used to compare both approaches. Performing a Welch Two Sample t-test on the two measurement series, yields, that they differ from each other (significance level 0.99). In other words, it is most likely, that the variability-aware interpreter is actually faster than the brute-force approach, which also rejects the null hypothesis for this third benchmark.

All performed empirical evaluations show, that the variability-aware interpreter is slower when interpreting code with few variability contexts. This outcome is conditioned by the variability checks, that are invoked by the variability-aware interpreter. For example, in every while loop iteration, it must be calculated, in which variability context the condition yields *true*. This also explains, why the variability-aware approach does not outperform the brute-force analysis at a lower number of features in the late splitting benchmark. Compared to a simple addition calculation, it takes more time to query the variability context of when the *while* condition yields *true*, which is not required in the plain interpreter. With a growing number of features, and thus an increasing number of variants, this initial penalty is compensated.

However, this work addresses the problem that brute-force analysis scale poorly for *high* numbers of features. Variability-aware interpretation is primarily designed for providing a suitable approach, when brute-force analysis can not yield a result in a reasonable amount of time. Therefore, performance problems, that only apply on very small product lines are negligible and in summary it can be stated, that variability-aware interpretation outperforms the brute-force approach in total.

4.6.1 Threats to validity

As every evaluation this work's empirical study must confront certain threats to validity. Possible threats can be divided into threats to internal validity and threats to external validity. While threats to internal validity represent factors that would provide alternative explanations for the benchmark results presented in Section 4.5, threats to external validity concern the generalizability of the variability-aware interpretation approach. In this section both types are discussed.

Internal validity. Circumstances that would influence the benchmark results are possible threats to internal validity. This includes an alternative explanation for the speedup values of the variability-aware interpreter. Such a threat is the impact of confounding factors in a benchmark. Examples for confounding factors are background processes, the execution of benchmarks on different systems or settings, and in some cases garbage collection. As described in Section 4.4, this evaluation has reacted to these threats by using the same hardware setup and java virtual machine settings for every benchmark, as well as closing every background process, which has not been needed for the benchmark to be carried out. Additionally, the garbage collector has been invoked manually before every benchmark. To control the impact of outliers, a warmup run has been executed before the actual benchmarks and all benchmarks describe the mean of three runs.

External validity. Factors that limit the generalizability of the approach presented in this work are threats of external validity. In the first two benchmarks of our evaluation we investigated favorable cases of the variability-aware interpreter. This may pose a possible threat to external validity because a speedup was already expected beforehand. However, these benchmarks served for providing some actual values for these cases, while the intention of the third benchmark was to provide a representative result. In this case, a possible threat to external validity would be, that the generated test product lines were not suitable as a set of subjects. For controlling this threat, a special test case generator, ScalaCheck [Nilsson], has been used to cover a variety of different cases concerning the structure of the subjects, such as empty product lines or product lines with excessive use a certain structure, e.g. many statements of the same type. Thus, a great variety of test cases is covered, and the set of subjects is stated as sufficiently representative.

Chapter 5

Future Work

The variability-aware interpreter presented in this work serves as a prototype, showing that applying the principles of variability-aware analysis on interpreting source code is an approach worth further research. However, future work concerning the interpreter mainly focuses on extending the supported language constructs. The used While language served well for showing, that variability-aware interpretation actually works, but in the presence of today's modern object oriented programming languages, it can not provide a practicable solution.

There is already an experimental branch of the variability-aware interpreter, that tries to add structures for supporting class definitions and objects. The extended interpreter is already able to handle simple class definitions as shown in Listing 5.1. Concerning object orientation, another store for class definitions is used and objects are treated as subclasses of type Value. Representing variability inside of classes, e.g. optional method definitions, is done by incorporating variability into these values, which technically means, that object values have their own variability-aware store and function store.

```
class Mult {
  const FACTOR = 2;

  def calc(x) {
    res = this.FACTOR * x;
  }
}
o = new Mult();
v = o.calc(3);
```

Listing 5.1: Object orientation for the variability-aware interpreter.

The difficulty in supporting objects is the great number of locations, where variability can appear. In this work, it has already been explained, that it is desirable to store variability as local as possible. However, this also implies great complexity. For example, classes can have variable fields, variable constructors and variable functions, which in turn can have optional parameters or optional statements in their bodies. Therefore, future work would primarily aim at tailoring together all these different variability contexts when interpreting programs. Tailoring together in this case means, that all locations, where variability can be specified have to be taken into account, when an object is accessed. For example, invoking the method of an object requires to handle the variability context of the object itself, of the function declaration, of possibly referenced variables and of the method parameters. As a result, future evaluations may also be able to take existing product lines, e.g. the Graph Product Line, as subjects.

Chapter 6

Related work

Software product-line testing. Testing is a first class concern in software development and a common method for maintaining the quality of software products. However, testing software product lines brings up the problem, that all variants of a product line have to be tested, to receive an exhaustive test result, which results in performance issues for product lines with high numbers of features.

One approach to address these performance issues, are sampling strategies [Cabral et al., 2010], which focus on reducing the number of variants under test, by selecting test cases by a specific selection criterion. Because sampling strategies can not provide exhaustive test results, when testing only a subset of all products, a different approach for handling the performance issues of brute-force analysis is needed.

As already explained in this work, variability-aware analysis provide a promising approach to address these performance issues. Developing a variability-aware interpreter, which has been done in this work, is one approach among others for product line testing, that is based on the principles of variability-aware analysis.

Parts of this thesis have already been published in a paper at the 4th International Workshop on Feature Oriented Software Development [Kästner et al., 2012]. In this paper, the white-box approach of developing a variability-aware interpreter from scratch is compared to using an existing model checker as black-box approach for product-line testing. On using the model checker, variability has been encoded with normal control-flow mechanisms, e.g. if statements. Kaestner et al. also introduce a "gray-box" strategy, which adds an extension to the standard model checker to improve its handling of variability, e.g. joining paths after variability related splits. Further references to related work concerning product-line testing can also be taken from the FOSD paper.

Variability-aware analysis. The variability-aware interpreter introduced in this work is an extension of the variability-aware analysis strategy, implemented to support interpreting source code. During its development we used parts of other approaches, that have presented strategies concerned to variability-aware analysis. For example, the TypeChef Parser [Kästner et al., 2011b], which provides the ability to parse variability-enriched code.

Other approaches related to variability-aware analysis concern type checking [Kästner et al., 2011a], static analysis of product lines [Kreutzer, 2012] and model checking [Lauenroth et al., 2009]. An overview of existing product-line analysis strategies is also available [Thüm et al., 2012]. In this work, variability-aware analysis are referred to as family-based analysis.

Symbolic execution. Another approach related to this work is symbolic execution [Cadar et al., 2011]. This technique uses symbolic values as input when executing programs and returns functions that describe the output of the program in relation to its symbolic inputs. When conditional statements occur, the values of program variables are maintained using a path condition for every possible path.

The variability-aware interpreter uses symbolic execution, as well as default concrete execution. Consider the following conditional statement: `if (a == 0) {...}`. When the evaluation of the condition yields a result, which is not dependent on a certain configuration concrete execution is used as the interpreter follows only one path, which is the actual result of evaluating the condition. However, if evaluating the condition does not provide the same result for all configurations, both branches are executed in the variability context that describes in which configuration the corresponding branch is entered, thus performing symbolic execution. Also, the variability context, in which each branch is executed can be compared to the path conditions in the symbolic execution approach.

Chapter 7

Conclusion

In this work, variability-aware interpretation has been presented as an alternative to traditional approaches of testing software product lines. First, the problems of existing approaches have been described. Brute-force analysis focus on testing all possible variants of a product line, thus analyzing common code over and over again, and running into performance issues for product lines with many features. Another approach, sampling strategies, reduces the number of tests, by selecting the tested variants with a special criterion. Though it solves performance issues, this approach is not able to provide a fully exhaustive test result and still focuses on testing variants of a product line.

Therefore, the concept of variability-aware analysis has been described and how it addresses the above-mentioned issues. Variability-aware analysis take variability into account when performing analysis on product lines, thus requiring only one process that analyzes common code once and provides a result for all configurations of a product-line. Also, abstract syntax trees have been introduced as a structure, where variability-aware analysis can be performed on. It was further shown, how variability can be injected into ASTs and how using the TypeChef Parser, an existing library for parsing variability-enriched code, helps to transfer source code in its corresponding AST representation.

As the main contribution of this work, the prototype of a variability-aware interpreter has been presented, that applies the concept of variability-aware analysis on interpreting source code and is able to execute test cases for all configurations. The interpreter has been described conceptually, as well as its implementation has been shown. Also, it has been achieved to transfer benefits of variability-aware analysis to the interpreter. For example, it executes programs in a single variability context as long as possible, resulting in common code getting executed only once. The interpreter's accepted language is a simple WHILE language and it has been shown how variability is handled in all of its language constructs.

To measure the performance of the variability-aware interpreter, an appropriate method for comparing the variability-aware approach to the brute-force approach has been presented. For that reason, a plain interpreter was introduced, that ignores all variability and represents the traditional way of interpreting source-code. For comparing both approaches, three benchmarks have been conducted, two of them showing the potential of the variability-aware approach in favorable cases, and another one over a set of random generated product lines to provide a comprehensive comparison over a variety of subjects.

The benchmarks results have shown, that the variability-aware interpreter outperforms the brute-force approach in total, especially when late splitting and early joining can be exploited. Limitations of the variability-interpreter's performance come into view on interpreting very small product lines, or in general, when the effort needed for interpreting a certain code fragment is less than the effort needed for checking the variability context of the execution.

These results emphasize the viability of variability-aware interpretation concerning the used WHILE language. However, to become an established approach for software product-line testing, the support of object oriented programming is essential. The variability-aware interpreter has been developed as a white-box approach from scratch. Its implementation is extensible and offers the possibility to add further language constructs. Also, it has been shown that some steps to supporting object orientation have already been done in an experimental branch.

All in all it can be said, that variability-aware interpretation offers a promising alternative to traditional software product-line testing approaches, that is worth further research.

Bibliography

The aspectj project. http://www.eclipse.org/aspectj/. Accessed: 06.10.2012.

Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X.

Isis Cabral, Myra B. Cohen, and Gregg Rothermel. Improving the testing and testability of software product lines. In *Proceedings of the 14th International Conference on Software Product Lines*, SPLC'10, pages 241–255, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15578-2, 978-3-642-15578-9.

Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42206-4.

Markus Kreutzer. Statische Analyse von Produktlinien. Bachelor thesis, April 2012.

Christian Kästner. Software product lines. Lecture, http://www.uni-marburg.de/fb12/ps/teaching/ss11/spl, 2011. Accessed: 06.10.2012.

Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2011a.

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 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 805–824, New York, NY, USA, October 2011b. ACM Press. ISBN 978-1-4503-0940-0.

Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward variability-aware testing. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, FOSD '12, pages 1–8, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1309-4.

Martin Kuhlemann, Sven Apel, and Thomas Leich. Streamlining feature-oriented designs. In *Proceedings of the 6th International Conference on Software Composition*, SC'07, pages 168–175, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-77350-9, 978-3-540-77350-4.

Kim Lauenroth, Klaus Pohl, and Simon Toehning. Model checking of domain artifacts in product line engineering. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 269–280, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3891-4.

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 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 105–114, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6.

Rickard Nilsson. Scalacheck github repository. `http://github.com/rickynils/scalacheck`. Accessed: 06.10.2012.

Michael Olan. Unit testing: test early, test often. *J. Comput. Sci. Coll.*, 19(2):319–328, December 2003. ISSN 1937-4771.

Mark Staples and Derrick Hill. Experiences adopting software product line development without a product line architecture. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, APSEC '04, pages 176–183, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2245-9.

Thomas Thüm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake. Analysis strategies for software product lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, April 2012.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Marburg, den October 11, 2012

Jonas Pusch