

Masterarbeit im Studiengang  
Angewandte Informatik

Variability-Aware Analysis of  
C Source Code

Adam Krafczyk

245935

[adam.krafczyk@uni-hildesheim.de](mailto:adam.krafczyk@uni-hildesheim.de)

**Betreuer:**

Prof. Dr. Klaus Schmid  
Sascha El-Sharkawy (M.Sc.)



---

## Eigenständigkeitserklärung

### **Erklärung über das selbstständige Verfassen von "Variability-Aware Analysis of C Source Code"**

Ich versichere hiermit, dass ich die vorstehende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der obigen Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich in jedem Fall durch die Angabe der Quelle bzw. der Herkunft, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet und anderen elektronischen Text- und Datensammlungen und dergleichen. Die eingereichte Arbeit ist nicht anderweitig als Prüfungsleistung verwendet worden oder in deutscher oder einer anderen Sprache als Veröffentlichung erschienen. Mir ist bewusst, dass wahrheitswidrige Angaben als Täuschung behandelt werden.

Hildesheim, den 27.11.2019

---

Adam Krafczyk

---



---

## Abstract

Variability in software projects is commonly used to create multiple variants of a software product. Especially software product lines make systematic use of variability to generate whole families of products. A popular mechanism to implement variability in C-code is conditional compilation via the C-preprocessor. Variability introduces new challenges that may be addressed with static analysis. However, traditional static analyses usually cannot handle variability and commonly ignore the C-preprocessor.

This thesis aims at enabling different static analysis approaches for C-source code containing C-preprocessor variability. The goal is to extract and represent the variability in the source code so that static analysis is possible. Two different approaches for static analysis of source code are developed. The first one analyzes only the C-preprocessor variability and extracts implicit domain knowledge from the implementation artifacts. The second approach is a parser that builds a combined model of the syntactical structure of the source code and the C-preprocessor variability. Both approaches are evaluated on real-world software product lines.

The first approach successfully extracts dependencies between variability variables from the source code assets. These dependencies represent domain knowledge that can be used to build variability model constraints. The second approach parses code into a code model that closely reflects the developer perspective on the content of the parsed source file, as opposed to the more abstract representations commonly used in static analyses. It enables analyses that require a combined view of the syntactical structure of the code and the variability.

---



# Contents

<b>List of Figures</b>	<b>ii</b>
<b>List of Tables</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>3</b>
2.1 Variability in Source Code . . . . .	3
2.2 Software Product Lines . . . . .	6
2.3 Static Analysis of Variability . . . . .	9
2.4 KernelHaven Analysis Framework . . . . .	12
<b>3 Analysis of Integer-Based C-preprocessor Variability</b>	<b>15</b>
3.1 Use-Case . . . . .	16
3.2 Non-Boolean Transformation . . . . .	20
3.3 Extraction of C-preprocessor Conditions . . . . .	28
3.4 Feature-Effect Analysis . . . . .	32
3.5 Result Simplification . . . . .	36
3.6 Analysis Results . . . . .	43
<b>4 Extraction of Source Code Combined with Variability</b>	<b>47</b>
4.1 Use-Case . . . . .	47
4.2 srcML Parsing Tool . . . . .	50
4.3 Transformation of srcML . . . . .	53
4.4 Alternative Approaches . . . . .	62
<b>5 Evaluation</b>	<b>66</b>
5.1 Non-Boolean Transformation . . . . .	66
5.2 CodeBlockExtractor . . . . .	72
5.3 Formula Simplification . . . . .	76
5.4 srcML-Extractor . . . . .	84
5.5 Threats to Validity . . . . .	88
<b>6 Conclusion</b>	<b>93</b>
<b>Bibliography</b>	<b>97</b>

## List of Figures

2.1	Example of C-preprocessor . . . . .	4
2.2	Resolved variability of Figure 2.1 . . . . .	5
2.3	KernelHaven reference architecture . . . . .	12
3.1	KernelHaven pipeline in the Bosch PS-EC use-case . . . . .	15
3.2	Resolving variability granularity in the Bosch PS-EC product line . . . . .	17
3.3	Overview of the non-Boolean transformation in the analysis process . . . . .	21
3.4	Integer sub-expression in a variability condition . . . . .	23
3.5	Conditional compilation blocks using the C-preprocessor . . . . .	29
3.6	Nesting structure of CodeBlocks . . . . .	30
3.7	C-preprocessor conditions for illustrating feature-effects . . . . .	33
3.8	Example Binary Decision Diagram . . . . .	38
3.9	Simplification steps in the feature-effect computation . . . . .	42
3.10	Ratio of dependent vs. independent variables per product variant . . . . .	44
3.11	Aggregated classification of independent vs. dependent variables . . . . .	44
3.12	Classification of variables grouped by number of products . . . . .	45
4.1	Example of C-source code parsed by srcML . . . . .	50
4.2	Formatted parsing result of srcML . . . . .	51
4.3	Example of C-source code with C-preprocessor directives parsed by srcML . . . . .	52
4.4	srcML-Extractor transformation process . . . . .	53
4.5	C-preprocessor markup created by srcML (simplified) . . . . .	54
4.6	Required C-preprocessor nesting (simplified) . . . . .	54
4.7	Problematic C-preprocessor markup created by srcML (simplified) . . . . .	56
4.8	Code with changing structure depending on conditional compilation . . . . .	56
4.9	srcML markup of Figure 4.8a (simplified) . . . . .	57
4.10	Transformation result with reference node (simplified) . . . . .	57
4.11	Example C-preprocessor construct that cannot be transformed . . . . .	58
4.12	Example C-preprocessor construct that srcML marks-up incorrectly . . . . .	59
4.13	AST class diagram of srcML-extractor (simplified excerpt) . . . . .	60
4.14	Parsing approach of TypeChef . . . . .	62
4.15	Non-trivial lifting of C-preprocessor variability . . . . .	64
5.1	Runtime with varying number of conditions . . . . .	68
5.2	Euntime with varying number of allowed values . . . . .	69
5.3	Ratio of extracted condition types . . . . .	74
5.4	Ratio of dependent vs. independent variables for different simplifications . . . . .	81
5.5	Total number of operators and variables relative to Simple simplification . . . . .	82
5.6	Ratio of files parseable by the srcML-extractor . . . . .	86

## List of Tables

2.1	Example excerpt of vending machine variability model . . . . .	8
5.1	Non-Boolean transformation measurements for two product variants. . . . .	69
5.2	Runtime measurements of different simplification settings (variant 1) . . . .	79
5.3	Runtime measurements of different simplification settings (variant 2) . . . .	79
5.4	<i>AAS</i> setting-specific runtime measurements (both variants) . . . . .	80
5.5	Runtime of srcML-extractor transformation steps . . . . .	85



# 1 Introduction

In software development it is quite common to develop multiple *variants* of a software product at once. These *variants* differ in some, sometimes only small aspect. Common use-cases for this are optional features, that can be en- or disabled in the build process, or platform-specific adaptations, that for example create one variant for Linux and another one for Windows. The differences between such variants are called *variability*. If multiple *variants* are developed in the same source code, there are mechanisms that implement the *variability*: at some point, the code changes depending on which *variant* is created. This may be during the build process, or even later at runtime.

Software product lines systematically use *variability* to implement a wide range of products, sometimes referred to as a *product family*. The assets of the product line are used to build product variants that are tailored for specific customer needs. Often, a whole domain of software products is covered by a single software product line. This requires a large amount of *variability* that enables the flexibility in the product creation. Whereas traditional software development sporadically uses *variability* to create some *variants*, the software product line approach systematically uses and manages the *variability* as a key success factor.

A common implementation of *variability* in C-source code is realized with the C-preprocessor, as it is included in all C-compilers. It has directives for *conditional compilation*, that means that parts of the source code are only compiled under certain conditions. Most prominently, the `#ifdef` directive can be used for this. This mechanism allows the implementation of multiple different *variants* of source code in a single file. At compile-time, it is decided which *variant* is built. This thesis will focus on *variability* that is implemented in this way.

The advantage of *variability* in the source code is that multiple different *variants* can be developed at once with relatively little overhead. The alternative is to develop all *variants* separately, which almost always increases the required effort. Especially for *variants* that only differ slightly, the overhead of developing them separately is too high. *Variability* also enables the software product line approach: without *variability*, a software product line could not generate such a huge range of tailored products. The assets from the product line that are re-used in many *variants* have to be flexible enough to adapt to different customer needs.

But *variability* in the source code also introduces some new challenges. The complexity of the source code increases, as now multiple variants of the program are depicted at once. This can make it harder for developers to understand the source code. Another challenge arises from the combination of *variability*, for instance the interaction of optional features. One optional feature may require that another optional feature is also enabled, or two optional features may be mutually exclusive. This can add a layer of interdependencies to the *variability*, which is not obvious when just looking at the source code. This challenge also decreases the understandability of the source code.

Static analyses are used in software development to assist the development process. They are often used in practice to find defects in source code. Traditional static analysis approaches and tools do not consider *variability*. C-preprocessor directives are either resolved or ignored by the tools. This means that only a single *variant* without *variability* can be analyzed by these tools. In contrast to that, *variability-aware* static analyses are developed specifically for source code that contains *variability* [TAK+14]. They are often used to address the specific challenges that arise from the *variability*.

This thesis presents two static analysis approaches for source code containing *variability* in the form of C-preprocessor conditional compilation. The focus is on the extraction and representation of the relevant information from the source code files. The two presented approaches build different models of the analyzed code.

**Analysis of Variability** In this approach, the *variability* inside the source code files is analyzed. The goal is to extract implicit domain knowledge from the `#ifdef` directives in the source code. This addresses the challenge posed by the implicit layer of interdependencies in the *variability*. This analysis makes the interdependencies explicit, which can help developers to understand and manage the *variability* in the source code.

For this approach, the `#ifdef` *variability* information has to be extracted from the source code files. A parser is developed and implemented that reads the *variability* information from the C-preprocessor directives and converts it into a model suitable for static analysis. Further, an existing analysis approach that extracts the implicit domain knowledge from this model is adapted and implemented. The resulting tool-chain is applied in a real-world industrial use-case, the Bosch PS-EC product line.

**Extraction of Source Code Combined with Variability** This approach builds a model of the source code that combines the syntactical structure of the C-code with the *variability* directives from the C-preprocessor. Based on an existing parser that creates syntactical markup for C-code, an approach is developed and implemented that builds such a combined code model. This approach enables static analyses that consider the syntactical structure of the code and the variability, and the interaction of both. The use-case for this is a framework for implementing *variability-aware* metrics, i.e. analyses that quantify certain aspects of the source code, the variability, or both.

This thesis is structured in the following way: First, Chapter 2 introduces core concepts that are important background for this thesis. This includes the concept of variability, software product lines, and variability-aware static analysis. Chapter 3 describes the approach that analyzes the variability in the Bosch PS-EC product line. The different aspects of the extractor, transformation, and analysis are explained. Chapter 4 describes the extraction of source code combined with variability. This covers the underlying parsing tool and the approach that creates the combined code model. Chapter 5 evaluates the different implementations that are created in this thesis. Finally, Chapter 6 concludes this thesis.

## 2 Background

This chapter introduces core concepts that are the basis for this thesis. First, Section 2.1 introduces the concept of variability in source code. Here, the idea of developing multiple variations of a software product at once is explored. Section 2.2 introduces the concept of software product lines, which expand and systemize the idea of variability in software projects. Software product lines are the underlying objects of investigation in this thesis. Section 2.3 addresses how the general concept of static analysis applies to a software project containing variability. Different concrete static analysis approaches for software product lines will be explored in this thesis. Finally, Section 2.4 introduces the KernelHaven framework for static analysis of software product lines. This framework is the basis for the implementations that are developed in this thesis.

### 2.1 Variability in Source Code

In software development it is quite common to develop multiple variations of a product at once. The resulting variants are basically the same product, but they contain slight differences that change the product in certain aspects. For example, the following use-cases for having multiple variations occur in practice:

- Platform-specific differences can be addressed, which allows the product to run on different target machines. For example, different variable data types may be needed on a 32-bit system compared to a 64-bit system.
- Some features of the product may be optional during the building process that compiles the product. For example, certain features may only be available if an optional library is available. If the library is missing, fallback routines are compiled into the product instead.
- Run-time switches can be used to modify the execution logic of the product. For example, consider the `ls` command on Unix systems. By default, it creates a space-separated list of the contents of a directory. However, when the `-l` switch is passed to it, it lists each entry on a single line and shows additional information about the files.

Such variations in a software engineering project are called *variability*. Through variability, multiple different *variants* of the product exist in a single source tree. Thus, a single project contains more than one product. Many aspects of the software are not variable and exist in all variants. However, the variable parts can lead to a whole range of slightly (or even moderately) different products. A range of related products that differ in some aspects are sometimes referred to as a *product family*.

The variability in a project needs to be resolved to create a runnable program. For each variable part of the source code (i.e. parts that have more than one possible variant) it needs to be determined which of the alternative variants should be used. Resolving

variability can happen during different stages of the development and build process. This is referred to as the *binding time* [VSR07, p. 12]:

**Checkout-time** variability is resolved when retrieving the source from a repository. For example, alternative source code files may be selected here.

**Compile-time** variability is resolved when (or directly before) the source code is compiled. Typically, this involves text-based transformations on the source code.

**Link-time** variability is resolved when the linker combines multiple objects into an executable binary. For example, this process decides which version of a library to use.

**Run-time** variability is resolved while the program is running. This variability is typically implemented via the standard functions of the programming language and control structures are used to evaluate which alternative to execute.

The variability in a project is typically implemented via Boolean expressions for the variable parts. The variant for which the *variability expression* evaluates to true will be used in the product. The variants which evaluate to false are discarded. The variables that are used in these variability expressions are called *variability variables*. They control which overall variant of the product is being built. A set of values for the variability variables is called a *configuration*. The configuration of the variability variables needs to be available at the binding time of the variability, that is when the variability is resolved. For instance, to resolve compile-time variability, a configuration is passed to the build system. The values for the variability variables are then used to resolve the variability expressions during the compilation process.

This thesis will focus on compile-time variability in projects written in the C-programming language. This kind of variability is typically realized by using the C-preprocessor. The C-preprocessor is implemented by all C-compilers, which means that no additional tooling is required for this variability implementation. It runs before the source code is parsed by the compiler. It evaluates directives starting with a hash character (#) and does text-based modifications on the source code. Some of these commands can be used for *conditional compilation*, which is used to implement variability. Additionally, the C-preprocessor also does macro expansion, which can be used to insert text snippets into the source code. Configurations for the variability variables can be passed either as command-line options to the compiler call (via the -D switch) or inside a header file using **#define** statements.

```
1 int calc(int a) {  
2     a = a * a;  
3     #ifdef CONFIG_MULT  
4         return CONFIG_FACTOR * a;  
5     #else  
6         return a / CONFIG_FACTOR;  
7     #endif  
8 }
```

Figure 2.1: Example of C-preprocessor

Figure 2.1 shows an example of how the C-preprocessor can be used for implementing variability via conditional compilation. The **#ifdef** directive in line 3 is used to select between two alternative implementations: If the variability variable **CONFIG\_MULT** is defined, line 4 is compiled and line 6 is discarded. Otherwise, line 4 is discarded and line 6

is compiled. This allows the user who compiles the program to switch between two alternative implementations, based on the configuration of the variable `CONFIG_MULT` that is passed to the compilation process. Additionally, `CONFIG_FACTOR`, used in lines 4 and 6, is a C-preprocessor macro; the content of that variable will be inserted in its place.

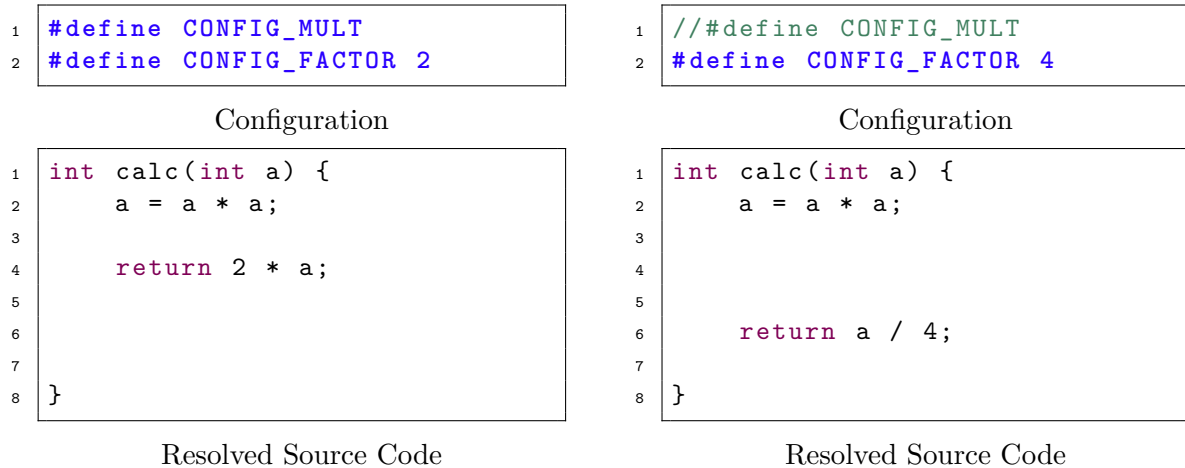


Figure 2.2: Resolved variability of Figure 2.1

Figure 2.2 shows two examples of how the variability in Figure 2.1 can be resolved. The figures at the top show configurations and the figures at the bottom show the corresponding source code with resolved variability. The configuration for the left example, defines the variable `CONFIG_MULT` (line 1) and sets the value 2 for `CONFIG_FACTOR`. This means that in the source code, the statement in line 4 in the `#ifdef`-block is selected, and `CONFIG_FACTOR` is replaced by the literal 2. This resolved source code is then passed to the compiler. The right example does not define `CONFIG_MULT` (line 1 is commented-out) and sets `CONFIG_FACTOR` to 4. This results in the `#else`-block to be selected, with `CONFIG_FACTOR` replaced by 4.

Using variability in the source code to manage multiple variants has the advantage of having relatively little overhead. Slight modifications only require a few extra lines of code. The alternative to in-source variability is to have multiple forks of the program, one for each variation. This clone-and-own approach requires the whole source code to be copied for each variation that is created. This alternative has several drawbacks:

- The common parts, i.e. the source code that is not changed over the different variations, may get out of sync. Changes in one variant are not automatically applied to other variants that have the same source code. In contrast, when using variability, the common source code is only stored in a single location. Changes to code outside of `#ifdef` blocks automatically applies to all variants.
- The clone-and-own approach also does not make it clear, what parts actually differ between the variants. Since whole source files are copied and then changed, an external diff-tool would be needed to show the differences between two variants. In contrast, with in-source variability, it is clear which parts change for which variant. `#ifdef` blocks are clearly visible when looking through a source file.

Despite all these advantages, using the C-preprocessor to implement compile-time variability also has a few challenges. First, it increases the complexity of the source code files. Source code files now do not only display a single program, but multiple variations of a program at once. This increases the visual complexity: a developer that looks that the

source code has to see all variations at once. This makes it harder to parse the actual structure of the underlying program. Additionally, the logical complexity of the source code is increased. Modifications to the common source code now have to address all possible variations at once. Modifications of the variable source code have to consider how and under which configurations these parts integrate into the product. This problem of increased complexity through conditional-compilation with the C-preprocessor is known under the term “Ifdef-Hell” [MRG+17].

Another challenge with variability in source code is that variability variables may have complex dependencies between them. Not all possible configurations for the variability variables may lead to a sensible program. For example, the face-recognition feature of a smartphone requires that a camera is present. A configuration where the face-recognition is enabled, but no camera is available, does not make sense. An `#ifdef FACE_RECOGNITION` implies an `#ifdef CAMERA`, but this implication is never stated in the source code. Such constraints add an implicit layer of complexity to all `#ifdef` blocks. However, for the developer to understand the code, knowing these dependencies may be crucial.

## 2.2 Software Product Lines

Software product line engineering is an approach to create multiple similar products from a single project, the software product line. The derivation of new products from the software product line is guided by systematic re-use of assets [VSR07, p. 6]. Assets in the product line are all artifacts relevant to a software engineering project: requirements, source code, tests, documentation, etc. The product line can be thought of as a construction kit for new products. It is aimed at a certain family of products, but still supports adapting each product to its specific needs.

As an example, consider an imaginary software product line for vending machines. The product line contains components for possible features of vending machines: payment (coins, banknotes, credit cards), user input (buttons, touchscreen), output (robotic arm, ticket printing, trapdoor), refrigeration, heating, etc. To create a new product, i.e. a new kind of vending machine, the assets implementing the desired features are taken from the product line. In the ideal case, when all required functionality is already present in the product line, this creates the complete product. However, sometimes product-specific adaptations may be required.

The benefit of software product lines is to reduce the *cost*, improve the *quality*, and reduce the *time-to-market* of products, while still allowing the creation of *taylor-made* products [ABK+13, p. 9f.]. The *cost* and *time-to-market* are reduced since re-use of existing assets requires significantly less effort than creating these assets from scratch. However, there is an up-front investment required to create the re-usable assets in the product line. This initial investment will be recapitalized over time by the lower effort of creating multiple products. Usually, the break-even point is reached at about 3 products [VSR07, p. 4]. The *quality* of products derived from a software product line is expected to increase, as the components used for creating the products are re-used across many products. Effort invested in improving the quality of product line assets will apply to all products that use these assets.

In the example of the vending machine, the imaginary developing company decided to create a software product line for all their vending machines. Initially, there was an increased

investment required to create the initial product line. It was developed by combining components from previously developed single-system vending machines into a new project covering the whole vending machine family. After that, many vending machines were created from the product line. These were both cheaper and faster to create compared to the competitors, which quickly offset the cost for the initial investment. Additionally, the quality of the software improved, as bug-fixes in the product line now get applied to many products at once.

Software product line engineering requires a systematic approach for developing re-usable assets and for re-using these assets to create products. It distinguishes between the following activities [VSR07, p. 7f.] [ABK+13, p. 20ff.]:

**Domain engineering** develops assets *for re-use* in multiple products. This creates a platform of re-usable elements that aims at supporting the products that will be derived from it in the future.

**Application engineering** develops products *with re-use* of the assets of the product line. A large part of the product can usually be created from the re-used assets. However, some product-specific development may be necessary to implement requirements that are specific to a single product.

The imaginary vending machine company has two departments responsible for product line development and product derivation. One is responsible to develop the product line platform (domain engineering). The market trends are analyzed to determine future developments in the field of vending machines. New features are then integrated into the product line, to make them available to future products. The other department derives products from the product line (application engineering). They elicit the requirements from the customers and then configure a fitting project from the product line. The product is quickly build from the re-usable parts of the product line. Sometimes, product-specific features need to be implemented, because they are not available in the product line. These product-specific features may get integrated into the product line platform at a later point.

A software product line contains a whole family of products. This means, that the assets that make up the product line inherently contain variability. The systematic management of this variability is an important aspect of software product line engineering [VSR07, p. 8ff.]. Elements of a software product line can be categorized in three types:

**Commonality** The element is present across all products derived from the product line.

**Variability** The element is present in some, but not all products.

**Product-specific** The element is specific to a single product.

In the imaginary vending machine product line, these three types of elements occur, for example, as follows: All vending machines have a maintenance mechanism, which is a commonality of all products. This includes an internal user-interface that allows maintenance activities of the different parts of the vending machine. A variability in the product line is the LCD monitor driver. It is present in some, but not all of the vending machines created from this product line. Finally, there are individual vending machines that have features that no other vending machine shares. For example, one machine has a key-card reader that handles payment through special key-cards that the users of that machine have.

The variability of a software product line is described by a *variability model*. It contains the set of all *variability variables* that describe the possible variations when creating a

new product. A *configuration* is a mapping of variability variables to concrete values. For each variability variable, it defines which alternative is selected. This way, the configuration describes a product that can be created from the product line. Additionally, the variability model defines constraints between the variability variables. This specifies, that not all possible combinations of values for variability variables describe sensible products. When creating a configuration for a new product, it is checked whether the desired configuration fulfills all constraints of the variability model.

Variable	Constraint
REFRIGERATION	$\text{REFRIGERATION} \implies \neg \text{HEATING}$
REFRIGERATION_TEMPERATURE	$\text{REFRIGERATION\_TEMPERATURE} \iff \text{REFRIGERATION}$ $\text{REFRIGERATION\_TEMPERATURE} < 20^\circ C$
HEATING	$\text{HEATING} \implies \neg \text{REFRIGERATION}$

Table 2.1: Example excerpt of vending machine variability model

An excerpt of the variability model of the imaginary vending machine product line is shown in Table 2.1. It contains these variability variables: **REFRIGERATION** is a Boolean variable describing whether there is a refrigeration unit for the stored items; **REFRIGERATION\_TEMPERATURE** is a floating point number defining the desired refrigeration temperature; **HEATING** is a Boolean variable describing whether there is a heating unit for the stored items. One constraint for these variables is, that **REFRIGERATION** and **HEATING** are mutually exclusive; heating and refrigeration at the same time does not make sense for a vending machine. Additionally, **REFRIGERATION\_TEMPERATURE** must be lower than room temperature, and must be set when **REFRIGERATION** is true; defining a desired cooling temperature without a refrigeration unit, or higher than the (expected) room temperature would be nonsensical. When a new vending machine is created from the product line, concrete values are selected for these variables. Then, it is checked whether all constraints for these concrete values are fulfilled. If this is not the case, the configuration needs to be changed in order to describe a valid product that can be created from the product line.

On the implementation side, variability in a software product line must be realized in the assets. The architecture of a product line supports the presence of variability [VSR07, p. 40ff.]. *Replacements* and *extensions* of components allow variability on an architectural scale. Variations are developed as separate components against a common interface or specification. This allows to resolve variability on larger scale, which moves complexity out of the different components of the architecture. However, variability may also be realized as *adaptations* inside the components. This is where explicit variability in source code is used, as described in Section 2.1.

In the imaginary vending machine product line, broader variability is handled in the architecture. For example, the different user input alternatives are implemented as separate components against a common user input interface. One component implements a touchscreen monitor, while another implements a keypad. This hides the complexity of variability to a certain degree, as all components just need to interact with the common interface. In contrast, smaller variability, like variations in the way the touchscreen works, are implemented inside these components. This is realized by using conditional-compilation of source code parts. This kind of variability appears throughout the source code and developers need to consider it when modifying the product line assets.

A software product line inherently contains variability, which results in challenges as described in Section 2.1. Variability increases the complexity of software projects and there

is a layer of interdependencies introduced by the variability variables. The systematic approach of product lines regarding variability helps to address these problems. Handling variability on an architectural scale helps to move the complexity out of the component implementations. Explicitly modeling the dependencies between variability variables helps to make that aspect more visible to developers. However, software product lines usually contain more variability than other software projects, as their primary intent is to cover a wider range of products. This makes it harder to deal with these challenges. While some variability is moved into the architecture, a software product line typically still contains a larger amount of conditional compilation in its source code, compared to other software projects. Dependencies between variability variables are explicitly modeled, but this typically results in complex models. Additionally, these dependencies are still not visible when just looking at the source code.

## 2.3 Static Analysis of Variability

*Static analysis* refers to automated analysis of software, without executing the software. It analyzes quality characteristics of the software, often with the goal to identify faults. This analysis is done statically on the source code or the compiled program. In contrast to that, *dynamic analysis* is performed by executing the program to analyze. Typical examples of static analyses are the detection of null-pointer accesses, out-of-bounds access of arrays, etc. In practice, static analysis is applied in software engineering to systematically search for defects. It complements other fault-detection techniques, such as manual inspections and dynamic testing [ZWN+06].

In software product line engineering, applying existing static analysis approaches is often not straightforward. These approaches have mostly been developed for traditional single-system development. The inherent variability in software product line assets is often not compatible with traditional static analyses. Practically speaking, most static analysis tools are not built to handle conditional compilation in the software they analyze. However, the advantages promised by applying static analysis in software engineering are still desirable for software product lines. Thus, there is a need to adapt the existing, or create new static analysis methods for product lines.

For software product lines, there are three main categories of analysis [TAK+14]:

**Product-Based** approaches only analyze a single product at a time. A product is fully derived from the product line (i.e. all variability is resolved) and is then statically analyzed. This has the advantage, that all existing static analysis tools created for single systems can be used without modifications. However, this approach can only find faults in the product configurations that it is executed on. Since many product lines have a vast amount of possible product configurations, applying this strategy is only feasible on a very small subset of products. A challenge here is the selection of which products should be analyzed to get the most coverage for the invested effort.

**Feature-Based** approaches analyze a single feature at a time. All assets that belong to a single feature representing a variability are analyzed in isolation. This allows for easier analysis, as the scope is limited to a small number of assets at a time. For analyzed characteristics that are compositional with respect to the features, this approach produces results for the whole product line. However, many characteristics

are dependent on interactions between features. These interactions, though, can not be analyzed when only considering a single feature at a time.

**Family-Based** approaches analyze the product line assets including all variability. This requires that the analysis approach can handle the variability that is present in the assets. In practice, this means that existing static analysis approaches need to be either extended or new approaches need to be developed. When this is the case, these approaches promise results that hold for the whole product line, i.e. all possible variations at once. This is typically much cheaper than generating all possible products and analyzing them individually, as there may be a vast amount of possible product configurations.

This thesis is working in the context of family-based static analysis. While the other approaches have their own benefits, we believe that family-based approaches are promising with regard to benefit of the produced results in the engineering process. Family-based approaches are executed on, and their results are applicable for the product line as a whole, including all variability. This means that they can be applied early-on in the domain-engineering process and their results are relevant for many products that will be derived from the product line platform. Family-based approaches have the most complete picture of the product line, while the other approaches are (intentionally) limited.

There already exists a number of family-based static analysis approaches for product lines. The following paragraphs will introduce a few examples: dead and undead code analysis, type-checking, feature-effect analysis, configuration mismatch analysis, and metrics.

The **dead and undead code analysis** finds conditional-compilation blocks that are either never or always included [TLS+11]. A conditional-compilation block (e.g. a C-preprocessor block) is called *dead* if its variability condition can never be fulfilled, considering the constraints in the variability model. For instance, a block with the variability condition  $\text{CONFIG\_A} \wedge \text{CONFIG\_B}$  is dead, if the variability model specifies that  $\text{CONFIG\_A}$  and  $\text{CONFIG\_B}$  are mutually exclusive. A dead block can be removed from the product line, as it will never have an influence on the derived products. Similarly, a conditional-compilation block is called *undead* if its variability condition always evaluates to true, considering the constraints in the variability model. For instance, a block with the variability condition  $\neg \text{CONFIG\_A} \vee \text{CONFIG\_B}$  is undead, if the variability model specifies the constraint  $\text{CONFIG\_A} \Rightarrow \text{CONFIG\_B}$ . Such a block is not real variability; its variability condition can be removed and the block can become a commonality of the product line. These kinds of problems arise from a mismatch of the variability constraints in the product line assets and the variability model. The dead and undead code analysis help to locate and remove such misleading variability from the source code.

**Type-checking** verifies that the analyzed source code adheres to the rules of the type-system of the programming language. For statically typed languages, this is implemented by the compiler. However, for product lines the compiler only runs after the conditional-compilation variability has been resolved. Thus, it only checks the type-safety of a single product at a time. To solve this problem, family-based type-checking has been proposed, which type-checks all variants of the product line at once. The result of this approach is either that all possible products of a product line are type-safe, or it shows configurations where the resulting product violates a type-system rule.

The TypeChef tool implements such family-based type-checker for the C-programming language with C-preprocessor based conditional-compilation [KKH+10; KGR+11]. A core concept of this approach is to parse the source code into a variable abstract syntax tree

(AST), which includes all variations at once. All elements in this AST have a presence condition annotated, which specifies under which variability condition they are included. The type-checker then builds a table of function and variable declarations, including their types and the presence condition of the declaration. For function calls and variable references, it is then checked that no type-system rules are violated; that means, it is checked that the usage matches the previously collected declaration. This matching considers the presence condition of the declaration, the presence condition of the usage, and the constraints of the variability model. This way, all valid combinations (i.e. configurations that fulfill the variability model constraints) of declaration and usage of variables and functions are checked at once.

The **feature-effect analysis** analyzes under which condition a variability variable has an *effect on the product derivation* [NBK+14]. An effect on the product derivation is assumed, when the variable controls whether a variability is included in the product or not. For conditional-compilation, this means that the variable controls whether the variability condition of a conditional-compilation block evaluates to true or false. As an example, consider a conditional-compilation block with the condition  $\text{CONFIG\_A} \wedge \text{CONFIG\_B}$ . If  $\text{CONFIG\_A}$  is true, toggling  $\text{CONFIG\_B}$  between true and false decides if the block is included or not. However, if  $\text{CONFIG\_A}$  is false, toggling  $\text{CONFIG\_B}$  has no effect anymore; the block is not included. Thus,  $\text{CONFIG\_B}$  only has an effect, if  $\text{CONFIG\_A}$  is true; the corresponding feature-effect is  $\text{CONFIG\_B} \Rightarrow \text{CONFIG\_A}$ . The feature-effect analysis calculates this for all variability conditions that appear in the product line; all cases where a variability variable occurs are considered for its feature-effect. The result of this shows for each variability variable, under which conditions it has an effect on the product derivation. This could be used to dynamically guide the configuration creation: variables that have no effect given the current partial configuration do not need to be configured anymore.

The **configuration mismatch analysis** analyzes mismatches between the variability in the product line assets and the variability model constraints [EKS17]. The basic idea of this analysis is to check if the feature-effects of the variability variables are included in the constraints of the variability model. A configuration mismatch is detected if the variability model constraints are more relaxed than the feature-effects. For instance, consider the feature-effect  $\text{CONFIG\_A} \Rightarrow (\text{CONFIG\_B} \wedge \text{CONFIG\_C})$  and the variability model constraint  $\text{CONFIG\_A} \Rightarrow \text{CONFIG\_B}$ . That is,  $\text{CONFIG\_A}$  can be selected in the variability model if  $\text{CONFIG\_B}$  is set, but  $\text{CONFIG\_A}$  has only an effect on the product derivation if  $\text{CONFIG\_B}$  and  $\text{CONFIG\_C}$  are set. In this example, the variability model is more relaxed than the feature-effect. This relaxed variability model results in two different, valid configurations leading to identical products: if  $\text{CONFIG\_B}$  is set and  $\text{CONFIG\_C}$  is not set, toggling  $\text{CONFIG\_A}$  has no effect on the product, but creates two different, valid configurations. This is a configuration mismatch; different configurations should always lead to different products. This problem can be addressed by either making the variability model more restrictive, or by making the feature-effect more relaxed, by handling more cases in the product line assets.

**Metrics** quantify different aspects of a software project. A common example here is the Lines of Code (LoC) metric, which counts the number of lines in the source code files. This can be used as an indicator for the size of a software project. *Variability-aware* metrics consider variability in their computation and are often defined specifically for software product lines [EYS18]. For example, the Number of Variation Points (NoVP) metric counts how many conditional compilation blocks are in the source code. There are also metrics defined for variability models. For instance, the Number of Features (NoF)

metric counts how many variability variables are in the variability model. Metrics can be used as a source of empirical data in a controlled software engineering process.

## 2.4 KernelHaven Analysis Framework

KernelHaven is a framework for statically analyzing software product lines [KES18b; KES18c]. Its goal is to enable writing analyses for product lines that are independent of the implementation details of specific product lines. To this end, KernelHaven defines a pipeline reference architecture that separates data extraction and analysis. Additionally, common utility tasks such as parallelization, writing output in different formats, logging, etc. are handled by the KernelHaven infrastructure. This way, analysis components can focus on their core task of implementing the analysis logic. KernelHaven is developed in Java and accessible as open-source [KE+19].

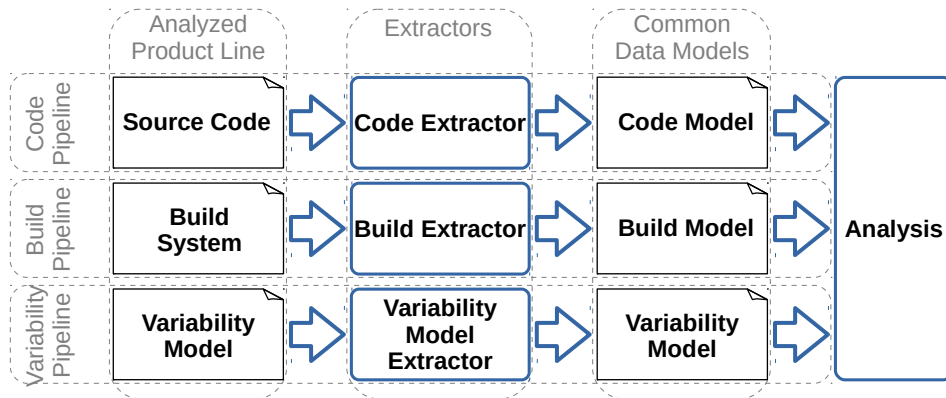


Figure 2.3: KernelHaven reference architecture

Figure 2.3 shows the reference architecture of KernelHaven. On the left are the assets of the software product line that is analyzed. The variability information from the assets is read by different extractors that parse it into common data models. The common data models are then used as the input for the analysis component. This approach makes the analysis component independent from implementation-specifics of the analyzed product line. The analysis only uses the common data models that are defined in the KernelHaven framework. These data models serve as a constant interface that does not change when analyzing different product lines. The extractors handle the implementation-specifics and transform the variability information read from the product line assets to the common data models.

KernelHaven has an extensive plug-in system that allows dynamic adaptation to different use-cases. The extractor and analysis components shown in the reference architecture in Figure 2.3 are exchangeable. The configuration created by the user, called the *pipeline configuration*, defines which extractors and analyses are used. The pipeline configuration is read at start-up time and the components are instantiated accordingly. Exchanging extractors allows adapting the analysis to different product lines. The analysis component can also be exchanged, which allows many different analyses to be conducted on a product line.

The plug-ins provide extractors, analyses, preparations, and general utility functions. Extractors and analyses are core components of the pipeline explained above. Preparations

are optional components that are run before the extractors start. They can be used to implement general adaptations of the analyzed product line. For instance, the analysis presented in Chapter 3 has a preparation that transforms the variability of the whole product line before the extractors parse it. Plug-ins can also provide general utility functions that are used by other plug-ins. This can range from simplification algorithms for Boolean formulas to input/output handlers for Excel worksheets.

The KernelHaven pipeline is split into three sub-pipelines (cf. Figure 2.3) that extract different kinds of variability information typically found in software product lines:

**Code Pipeline** This sub-pipeline extracts variability information from the source code files. Typically, variability in the source code involves conditional compilation blocks, for example implemented using the C-preprocessor. The KernelHaven data model for this sub-pipeline is generic and allows for extensions, as there is a variety of different variability information that may be extracted here.

**Build Pipeline** This sub-pipeline extracts variability information from the build system. The build system typically contains file-level presence conditions for the source code files. The KernelHaven data model for this sub-pipeline is a simple mapping of files and presence conditions.

**Variability Model Pipeline** This sub-pipeline extracts variability information from the variability model. The variability model typically contain a list of all variability variables and constraints between these variables. The KernelHaven data model for this sub-pipeline reflects these two parts of a variability model. The variability variables are collected in a set and are referenced by their name. The data stored for each variable is extensible to support the variety of different variable types. A data file contains the constraints of the variability model. This is typically stored in the DIMACS format, which contains propositional formulas in conjunctive normal form. However, other formats are possible, which allows the model to be adapted to different kinds of variability models.

In practice, all variability information in a product line can be extracted by one of these sub-pipelines. However, not all product lines may have information for all three sub-pipelines, and not all analyses require the information of all three sub-pipelines. Thus, in a specific execution some sub-pipelines may be left empty. For example, a product line may not have a variability model. In this case, the variability model sub-pipeline is not used.

The analysis components can also be chained to form a processing pipeline. Many analyses can be broken down into several steps that the variability information is passed through. Some of these steps are also common in multiple analyses. The analysis steps can be implemented as independent analysis components. The full analysis approaches are then defined by combining the required components into a pipeline. For instance, the configuration mismatch analysis introduced in the previous section is an extension of the feature-effect analysis. The analysis components of the feature-effect analysis are re-used in the configuration mismatch analysis pipeline.

The properties of KernelHaven fulfill many of the requirements for an experimentation workbench for static software product line analysis [SEK19]. An experimentation workbench aims at supporting the process of creating and refining experiments in software engineering. In the context of static software product line analysis, experimentation means

that new analyses are implemented and evaluated. KernelHaven acts as an experimentation workbench, supporting the process in different aspects:

- The flexible analysis component structure of KernelHaven allows to easily implement new analysis ideas. This is further supported by re-using analysis components from existing, previously implemented analysis approaches.
- The decoupling of analyses from implementation details of the analyzed product lines allows for easy execution of existing analyses on different product lines. Analysis ideas only have to be implemented once using the common data model of KernelHaven. After that, they can be tested on many product lines by exchanging the corresponding extractors.
- The results produced by a KernelHaven analysis can be easily integrated into further processing and analysis frameworks. This supports the evaluation of experimentation results. This is enabled as KernelHaven supports many different output formats that can be changed transparently in the pipeline configuration. These output formats range from CSV, over Excel to SQLite databases, and can be further extended by additional plug-ins. For example, KernelHaven can be configured to output in CSV which can then easily be used for statistical analysis in the R programming language. Another example is in the ITEA project REVaMP<sup>2</sup>, where KernelHaven integrates into a broader toolchain [GBA+19]. One such integration is that the output of a feature-effect analysis by KernelHaven is written to an SQLite database, which is then visualized by the FeDeV tool.
- The pipeline configuration of KernelHaven serves as a description of the experiment. It contains the setup of the different components of the pipelines: the extractors and the processing pipeline of the analysis components are defined. Experiments can easily be shared and repeated by supplying the pipeline configuration and the necessary plug-ins.
- The process of developing a new analysis idea often involves rapid prototyping. Here, it is beneficial when the prototype can be tested on a product line without much overhead. KernelHaven supports this with the option to cache the results of the extractors. Some extractors have a long runtime, which would hinder the fast iterations of prototypes. The caching system allows the developer to test the analysis prototype on real data, without a significant runtime overhead. Additionally, different extractors can be used during testing and for the final analysis. In the testing phase, a less exact extractors that has a short runtime can be used, while for the final analysis a more exact but slower one can be selected.
- KernelHaven offers an option that archives all relevant assets of an analysis after it is finished. This includes the pipeline configuration, the plug-ins, the analysis output, and the analyzed product line. This supports the requirements of many researchers that the experiment should be archived.
- KernelHaven has a growing number of existing open-source plug-ins. These cover extractors for different product lines, analysis components, and utility functionality. This body of existing work can be easily re-used when developing new analyses.

### 3 Analysis of Integer-Based C-preprocessor Variability

This chapter presents a static analysis that was conducted in an industrial use-case. The objective of the analysis is to reverse-engineer knowledge that is implicitly encoded in the variability conditions of the Bosch PS-EC software product line. This involves two general steps: First, the variability information of the PS-EC product line needs to be extracted. This covers the variability model that defines all variability variables, the configuration management system that defines presence conditions for components and source code files, and conditional compilation blocks in the source code and other assets of the software product line. This step builds a model of the variability in the product line. Then, based on this model, the variability information of the product line is analyzed. This step aims at extracting domain knowledge, that is implicitly encoded in the variability conditions.

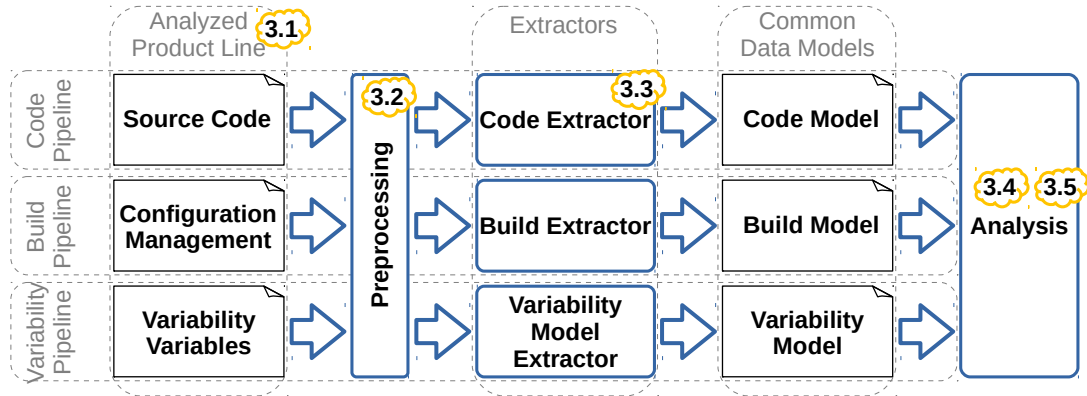


Figure 3.1: KernelHaven pipeline in the Bosch PS-EC use-case

The analysis presented in this chapter is implemented using the KernelHaven analysis framework (cf. Section 2.4). Figure 3.1 shows an overview of the adapted KernelHaven pipeline structure used for this analysis. On the left, the sources of variability information in the PS-EC product line are passed through a preprocessing step. The preprocessed product line assets are then parsed by the extractors of the three sub-pipelines. The resulting models are then passed to the analysis component.

The sections in this chapter cover different aspects of the analysis, in the order they appear in the analysis pipeline. The small clouds in Figure 3.1 indicate the sections in which the respective part of the analysis pipeline is covered. First, Section 3.1 introduces the industrial use-case where this analysis was conducted. It describes how the variability is implemented and what challenges arise when extracting the variability information. Section 3.2 introduces the preprocessing step that prepares the product line for analysis. It explains how the integer-based variability information can be encoded using pure Boolean formulas. Section 3.3 describes the code extractor that is used to extract C-preprocessor conditional compilation blocks from C-source code. Section 3.4 introduces the feature-effect analysis, that is used to extract the domain knowledge from the variability information. Section 3.5 discusses different techniques that are used to simplify the resulting Boolean formulas. Finally, Section 3.6 presents the results found with this analysis.

This chapter is largely based on previous work published in:

Sascha El-Sharkawy, Saura Jyoti Dhar, Adam Krafczyk, Slawomir Duszynski, Tobias Beichter, and Klaus Schmid. “Reverse Engineering Variability in an Industrial Product Line: Observations and Lessons Learned”. In: *Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC’18)*. Vol. 1. - ACM, 2018, pp. 215–225. DOI: 10.1145/3233027.3233047

### 3.1 Use-Case

The business unit Powertrain Solutions - Electronic Controls (PS-EC) of the Robert Bosch GmbH develops a software product line in the domain of embedded electronic control units (ECU) for diesel, gasoline, hybrid, and electric engines. ECUs control the functions of an engine, like fuel intake, torque, and battery charging [TBM+12]. They are used in a wide variety of applications, like scooters, motorbikes, passenger cars, delivery vehicles, trucks, construction machinery, and stationary industrial engines. The PS-EC product line is a well-known hall-of-fame member of the Software Product Line Conference [SHF07].

The PS-EC product line consists of 300 subsystems. These are independently managed by experts in the respective technical domain. Typically, 100 of these subsystems are included in a product. This amount of decomposition in the architecture allows for highly distributed development. The PS-EC product line is used to create, on average, 2000 new products per year.

The variability in the PS-EC product line is driven by many factors [STB+04]. Technical factors include the variability in the engine hardware, which the software needs to adapt to, and divergent concepts for the powertrain, like electric and hybrid vehicles. Organizational factors increasing the variability stem from different business models. These range from Bosch being a full service provider, supplying all hard- and software, to Bosch only supplying the hardware and device drivers. With globally acting customers, differences in legislation also need to be considered. Finally, customer-specific requirements need to be fulfilled, which allows Bosch to offer a competitive product portfolio.

Variability in the PS-EC product line is realized in different levels of granularity. A subsystem may have different alternative code modules as implementations. This allows for even higher distribution of development, as unrelated features in the same subsystem can be developed independently in different code modules. Additionally, conditional compilation of the C-preprocessor is used for more fine-grained variability. When creating a new product, the variability is resolved in two steps: First, it is selected which subsystems need to be included and which alternative implementation of these to use. Then, the C-preprocessor variability is resolved during the compilation process.

Figure 3.2 illustrates the steps in which the variability of the PS-EC product line is resolved. At the top, all variability is still present. There are three subsystems: A, B, and C. Subsystem B has two alternative implementations. The implementations of the subsystems A and B also contain C-preprocessor variability. In the first step, subsystems A and B are selected to be included in the product. Subsystem C is left out. Additionally, implementation 2 of subsystem B is selected over implementation 1. The second step resolves the C-preprocessor statements. The `#if` block in the implementation of subsystem A is included, while the one in the implementation of subsystem B is not.

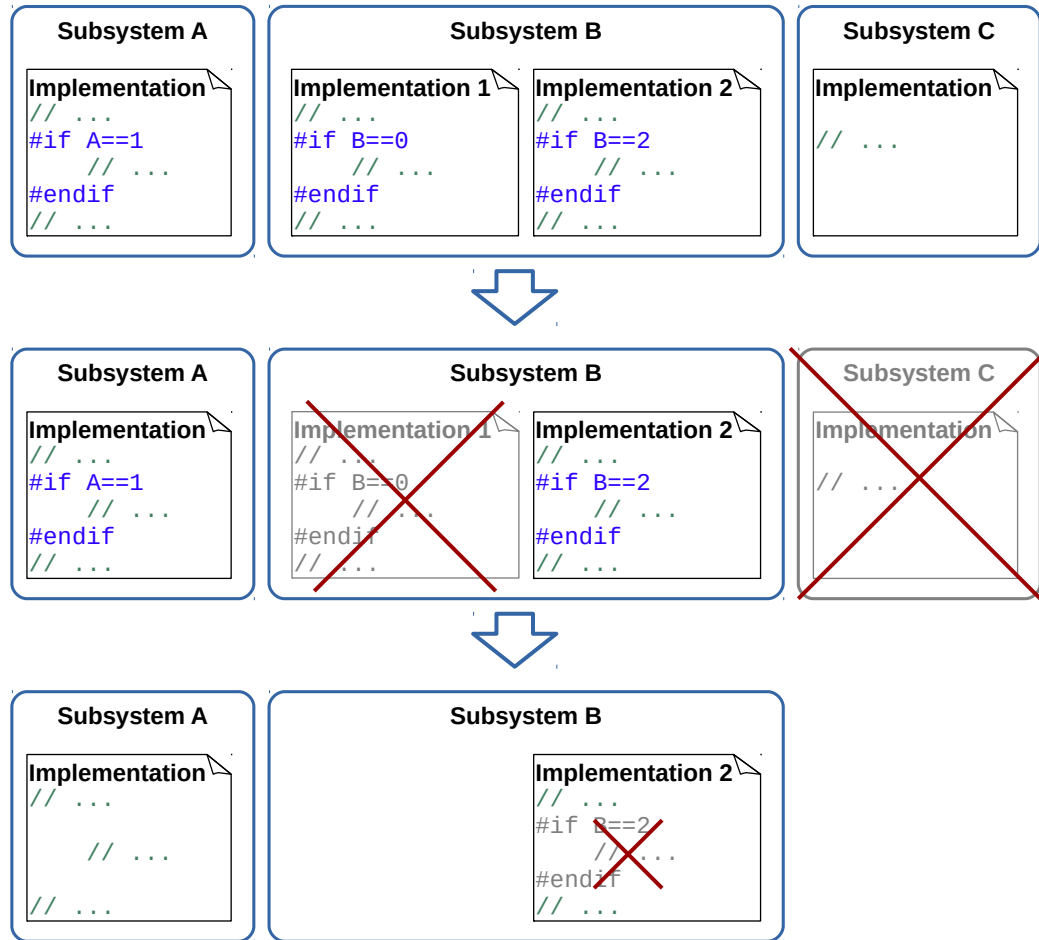


Figure 3.2: Resolving variability granularity in the Bosch PS-EC product line

The variability of the PS-EC product line is rigorously managed. All variability variables, their dependencies, and possible values are documented. Additionally, each variability variable is clearly owned by a single subsystem or implementation alternative. Special tools are employed to assist handling the variability. For instance, a configuration management system performs the process of selecting the subsystems and implementation alternatives (step 1 in Figure 3.2). The subsystems and alternative implementations have variability conditions that define their inclusion, which are automatically evaluated by the configuration management system based on the current product configuration.

The process to create a configuration for a new product, while partially automated, involves domain experts of the used subsystems. These experts use their domain knowledge to ensure that a valid configuration is created. That is, they ensure that the selected values for the variability variables do not conflict with each other. This is required, as the variability model does not contain formalized constraints to check the consistency of configurations.

The configuration process is an area where the PS-EC business unit wants to improve its product line. The effort required from domain experts can be reduced by formalizing domain knowledge in the variability model in the form of constraints between the variability variables. This way, non-sensible combinations of values for variability variables can be detected and excluded during the configuration process. Automating this process requires drastically reduced effort, compared to manually involving domain experts. Additionally,

an automated formalized approach reduces the risks inherent to manual configuration checking done by humans, for example by overlooking a detail. On the other hand, creating the required constraints in the first place is not trivial. It requires a comprehensive view on the variability that is implemented in the product line. The domain knowledge must be formalized in a way that covers all possible variants at once.

The goal of the analysis presented in this chapter is to help create an initial set of constraints for the variability variables. The idea is to reverse engineer implicit domain knowledge that is present in the already existing variability implementations. That is, the usage of variability variables in the existing variability conditions in the different artifacts of the product line is used to extract domain knowledge that is implicitly present. The resulting formulas can then be used as a starting point for the constraints in the variability model. Based on the extracted domain knowledge, the domain experts can manually expand the constraints. The reverse engineering should extract as much domain knowledge from the implementation as possible, to reduce the effort required from the domain experts in the following manual expansion. Additionally, the formulas created by the automated extraction should be as concise as possible. Concise formulas are often easier to understand and thus more suitable to build the variability model upon.

Creating constraints in a global variability model, covering the whole PS-EC product line, is not intended. With internationally distributed development of the many subsystem alternatives, it is simply impractical to try to keep a comprehensive model of all variability constraints in sync. Instead, the goal is to create constraints on a smaller, per-subsystem / per-alternative-implementation scale. This resembles more closely that each subsystem is developed and managed independently by different groups of technical domain experts. When creating a new product, the constraint models of the involved subsystems can be combined to a product-level variability model.

The PS-EC product line cannot be analyzed as a whole. This is because in the first step of resolving variability (subsystem and alternative selection, cf. Figure 3.2), a complete selection of the whole product line is not possible. Alternatives cannot be selected together, as they supply the same set of files, which leads to file collisions. Additionally, the subsystems have complex interdependencies, often to specific versions in time, which also cannot be selected at the same time.

For these reasons, the analysis in this chapter will not analyze the full PS-EC product line, but multiple *product variants* that are derived from it. This derivation is based on product configurations that were created for real-world customers. The product variants have the first level of variability granularity resolved, that is the subsystem and alternative selection has been performed. While the deselected alternatives and other subsystems are not available for analysis, the variability conditions of the selected subsystems and alternatives are still available. Additionally, the C-preprocessor variability conditions in the source code are also available for analysis. On the spectrum of a 150 % model representing the unresolved variability of the whole product line, and a 100 % model only representing the resolved variability in a completely configured product, we consider these partially resolved product variants to represent a 120 % model of variability.

The reverse engineering process considers the following sources of variability information from the PS-EC product line:

**List of variability variables** A list that defines all variability variables. It lists the name of the variable, the type, and the set of values that the variable is allowed to be assigned. All variable types are realized as integers. Some do not have restrictions on

the allowed values (unrestricted variables), while most have a smaller set of allowed values (like enumerations). For Boolean variables, the value 0 is assigned to signify *false*, while the value 1 signifies *true*. For all variables, there is also the option to not define them, that is, to not set any value for them. In KernelHaven, the information from this source is extracted in the variability pipeline.

**Conditions in the C-preprocessor** The source code contains C-preprocessor conditional compilation using variability variables. The resulting presence conditions for the source code blocks are extracted. In KernelHaven, the information from this source is extracted in the code pipeline.

**Conditions in the configuration management system** The configuration management system, that selects which subsystems and implementation alternatives are used in a product, contains variability conditions. These are used to automatically select the subsystems and implementation alternatives based on the product configuration. Thus, these variability conditions are used as presence conditions for whole subsystems and implementation alternatives. In KernelHaven, the information from this source is extracted in the build pipeline.

**Conditions in interface descriptions** The product line contains interface descriptions in XML files. These are used to automatically generate interfaces and follow industry standards like MSR, MDX, and AUTOSAR. Some of the interfaces in these files have variability conditions that define when they are provided. These are essentially equivalent to presence conditions like C-preprocessor blocks in source code. In KernelHaven, the information from this source is extracted in the code pipeline.

**List of legacy variability variables** Over time, some previous variability in the product line is no longer considered to be variable anymore; it has either been permanently excluded or included. To support legacy projects, the corresponding variability variables and variability conditions have not been removed from the implementation assets. Instead, a list contains all these legacy variability variables and defines fixed values for them. These are now always set as constants, effectively eliminating their variability for new products. In KernelHaven, the information from this source is extracted in the variability pipeline.

A challenge of the analysis in this chapter is to extract and combine these diverse sources of variability into one consistent view. Considering as many sources of variability as possible promises the most amount of domain knowledge to be reverse engineered, provided that it can be combined consistently for analysis. Another challenge posed by this product line is that the variability variables all hold integer values. Many of the existing analysis techniques and tools for C-preprocessor variability expect Boolean variability conditions. Expressions with integer variables and operators are mostly not considered. Thus, these approaches need to be modified so that they can be applied to the PS-EC product line.

## 3.2 Non-Boolean Transformation

This section is based on work previously published in:

Adam Krafczyk, Sascha El-Sharkawy, and Klaus Schmid. “Reverse engineering code dependencies: converting integer-based variability to propositional logic”. In: *Proceedings of the 22nd International Conference on Systems and Software Product Line-Volume 2*. ACM. 2018, pp. 34–41. DOI: 10.1145/3236405.3237202

The variability variables in the Bosch PS-EC product line hold integer values. Most of them represent Boolean variables (0 for false, 1 for true) or (small) enumerations of allowed values, while some are unrestricted (e.g. the whole 32-bit integer range is allowed). The existing variability model of the product line has a list of all the variability variables and their allowed values. The variability conditions in the C-preprocessor (and other places) use integer arithmetic operators (+, −, \*, /, %, &, |, ^, ~) and integer comparison operators (==, !=, <, <=, >, >=) on them. Additionally, the Boolean operators (&&, ||, !) and the `defined` function are used. The `defined` function returns whether *any* value has been set for a given variable. Variability conditions constructed with these operations define whether code blocks should be included in the compilation process or not (conditional compilation). For example, such a condition may look like this:

```
#if (VAR_A * 2 > VAR_B) || defined(VAR_C) (3.1)
```

Code following this C-preprocessor directive, until the next `#endif`, is only included in the compilation process if the value of `VAR_A` times 2 is greater than the value of `VAR_B` or if `VAR_C` is set to any value. The variability model defines the allowed values for the three variables, for example: `VAR_A` ∈ {1, 2, 3}, `VAR_B` ∈ {5, 6}, and `VAR_C` ∈ {0, 1}.

Other software product lines using the C-preprocessor that are commonly analyzed in academia, like the Linux Kernel [Lin19], usually have a different approach to how variability variables are handled. In them, Boolean variability variables<sup>1</sup> are predominantly used [BSL+13]. For example, in the Linux Kernel only 3% of the 6320 variability variables are integers, and 0.4% are string variables. The rest are Boolean variables (or at least implemented as Boolean variables in the C-preprocessor). In contrast to the PS-EC product line, Boolean variables in other C-preprocessor product lines are often not implemented by setting different values for a C-preprocessor variable. Instead, the distinction between the states true and false is implemented by either defining or not defining the C-preprocessor variable. The variability conditions then use the `defined` function (or the shorthand `#ifdef`) to check if a given variability variable is set.

Many of the family-based analysis techniques defined for software product lines use SAT-based approaches on propositional logic. For instance, the dead and undead code analysis, the feature-effect analysis, and the configuration mismatch analysis introduced in Chapter 2.3 fall into this category. They only work on variability conditions in propositional logic, with only Boolean variability variables. Such SAT-based approaches work well on the Linux Kernel, as the small number of non-Boolean variability variables can be ignored

---

<sup>1</sup>The Linux Kernel also has the concept of *tristate* variables. These are variables that allow three possible states. They are implemented as two Boolean variables in the C-preprocessor, and thus can be considered as Boolean variables from the perspective of an analysis tool.

while still creating reasonable results. However, these approaches and tools do not work on the integer-based variability of the PS-EC product line.

One possible approach to apply SAT-based analysis approaches on the PS-EC product line is to adapt the approaches to also handle the integer-based variability expressions. On the conceptual side, the approaches are currently only defined for variability conditions in propositional logic. They need to be expanded to handle higher-order logic expressions, like integer arithmetic and comparisons. For the feature-effect analysis, for example, this is not straightforward, as a core definition relies on setting variability variables to true and false. This cannot simply be applied to non-Boolean variability variables. In addition to the conceptual challenges, there are also technical problems to consider. The tools that implement the analysis approaches need to be expanded to be able to parse and represent integer-based formulas as present in the PS-EC product line. Additionally, the SAT-solvers that most of the tools employ need to be exchanged for solvers that can handle more than propositional logic. Because of these conceptual and technical challenges, significant effort is required to expand existing SAT-based approaches to be able to handle the integer-based variability of the PS-EC product line. Thus, a different approach that requires less effort was developed in the context of this thesis.

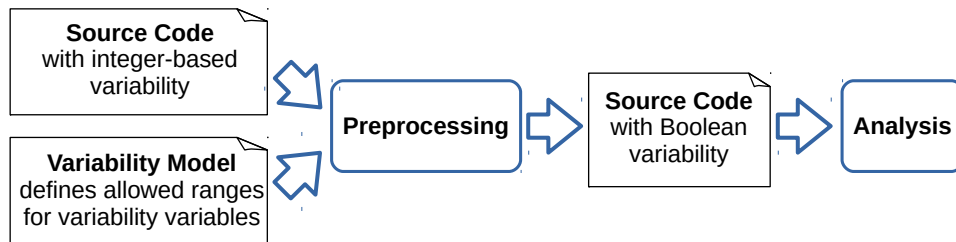


Figure 3.3: Overview of the non-Boolean transformation in the analysis process

The solution that was developed in the context of this thesis is to convert the integer-based variability to equisatisfiable propositional formulas. With this approach, all variability conditions are converted in a preprocessing step, before they are passed to the analysis. The result of the conversion are pure Boolean expressions, that hold the same variability information with respect to satisfiability as the original variability expressions. Thus, the existing SAT-based analysis approaches and tools can be used without modification to create useful results for the PS-EC product line. Figure 3.3 shows the preprocessing process: the integer-based variability in the source code is, based on the information from the variability model, transformed to propositional logic. This is then passed to the SAT-based analysis.

The `NonBooleanPreperation` class is implemented as a preprocessing component in the KernelHaven pipeline [KE18c]. Preprocessing components are executed before any extraction or analysis takes places. The non-Boolean preprocessing copies the whole source code tree while replacing all integer-based C-preprocessor conditions. This results in a source code tree with pure Boolean variability in the C-source files, which allows any of the existing Boolean-based extractors and analysis tools to be run. The preparation converts integer-based conditions in the following steps:

1. Parse the condition to an abstract syntax tree (AST)
2. Transform the AST so that it only contains Boolean operators and variables

3. Convert the transformed AST back into a C-preprocessor string to replace the original condition

The preparation component for KernelHaven applies this conversion automatically to all C-preprocessor conditions in source code files. In addition to that, the conversion approach is used by extractors that read integer-based variability and need to convert it to propositional logic. For instance, the extractors that read other variability sources of the PS-EC product line (e.g. the configuration management) use this conversion mechanism to convert their variability conditions.

The transformation step introduces Boolean variables for each possible value of the variability variables. This is feasible, because most of the variability variables have only a small range of allowed values. For each integer-based condition, the transformation calculates which combinations of values satisfy this condition. From this, a propositional formula using the introduced Boolean variables is created, that reflects these combinations. This approach ensures that the variability information that is important for the SAT-based analyses is preserved. In addition to the Boolean variables for each possible value, another Boolean variable is introduced for each variability variable, that denotes whether that variable is defined (i.e. whether it is set to *any* value). This is used in the transformation process to convert calls of the `defined` function of the C-preprocessor.

The mapping of allowed values per variability variable to the Boolean replacements can be formalized as follows:

Let  $V$  be the set of all integer-based variability variables,  $R : V \rightarrow \mathcal{P}(\mathbb{Z})$  a function that defines the range of allowed values for each variable, and  $B$  a set of Boolean variables. We then introduce a function

$$\sigma : V \times (\mathbb{Z} \cup \{\epsilon\}) \rightarrow B$$

which injectively maps a variable and one possible value of it to a Boolean variable.  $\epsilon$  in place of a value maps to the Boolean variable that denotes whether the integer variable is defined or not (i.e. whether it is set to *any* value). For instance, consider the variable `VAR_A` with  $R(\text{VAR\_A}) = \{1, 2, 3\}$ . `VAR_A` may be set to either 1, 2, or 3.  $\sigma(\text{VAR\_A}, \epsilon)$  returns the Boolean variable that denotes whether `VAR_A` is defined.  $\sigma(\text{VAR\_A}, 1)$  returns the Boolean variable that denotes that `VAR_A` is set to 1. In practice, such variable names may look like this:  $\sigma(\text{VAR\_A}, 1) = \text{VAR\_A\_eq\_1}$ ,  $\sigma(\text{VAR\_A}, \epsilon) = \text{VAR\_A}$ . The exact naming scheme depends on the context that the approach is applied in; it has to ensure that no name collisions occur, and that the names are valid identifiers for the C-preprocessor.

There are two constraints for the introduced Boolean variables, which are not explicitly modeled. These have to be manually considered when interpreting the result of any analysis done on the propositional formulas:

1. The Boolean variables for the possible values of an integer variable are mutually exclusive:

$$\forall v \in V, \forall i, j \in R(v), i \neq j \mid \sigma(v, i) \implies \neg \sigma(v, j)$$

2. The  $\epsilon$  Boolean variable is true if and only if a value is set for the integer variable, that is if any of the Boolean variables denoting the possible values is true:

$$\forall v \in V \mid \sigma(v, \epsilon) \iff \bigvee_{i \in R(v)} \sigma(v, i)$$

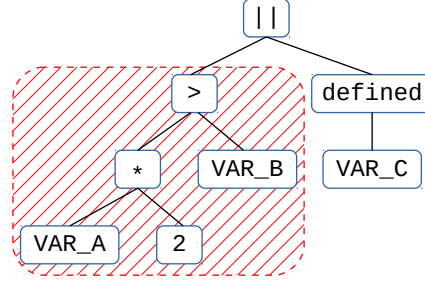


Figure 3.4: Integer sub-expression in a variability condition

The integer-based variability conditions are converted to propositional logic by replacing all integer sub-expressions inside of them. Figure 3.4 shows the abstract syntax tree (AST) of the variability condition shown in Formula 3.1. The integer sub-expression that must be replaced by a Boolean equivalent is highlighted. The highest operator of any integer sub-expression is always a comparison operator.<sup>2</sup> On the left and right side of this comparison, there are literals, variables, or arithmetic operations combining both. The general idea is to find all possible combinations of allowed values for the variables on the left and right side that fulfill the comparison operator. These values are then transformed into Boolean variables using the  $\sigma$  function. A propositional formula is constructed from them, that is satisfiable for all combinations that fulfill the comparison.

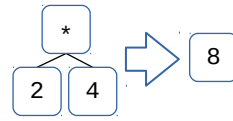
The integer parts of a variability condition are resolved by walking bottom-up through the AST. Arithmetic operations are applied on all allowed values of the variability variables at once. This results in a set of values, instead of a single result value for the operation. This is needed, because the transformation wants to find *all* possible values that fulfill the comparison at once. Eventually, when the comparison operator is reached, it is checked which *original* values of the variability variables fulfill the comparison operator. These *original* values are required for the  $\sigma$  function to get the Boolean replacement variable. Thus, when applying arithmetic operations, this approach keeps track which *original* value of the variability variable led to the *current* result of the arithmetic operation. For instance, when resolving the arithmetic operation `VAR_A * 2`, with  $R(\text{VAR\_A}) = \{1, 2, 3\}$ , the following tuples of *current* and *original* values are computed:  $\{(2, 1), (4, 2), (6, 3)\}$ .

The following rules are used to evaluate the integer parts of the AST. They are applied based on which integer operator is used on which input types.

- *comparison operator* refers to integer comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`)
- *arithmetic operator* refers to integer arithmetic operators (`+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `~`)
- *literal* refers to literal integer values
- *variable* refers to integer variables, with a defined range of allowed values

<sup>2</sup>Sometimes there are no explicit comparison operators to convert integer expressions to Boolean values. In this case, a `!= 0` comparison can be assumed, since all integer values except 0 are defined to be `true` in the C-preprocessor.

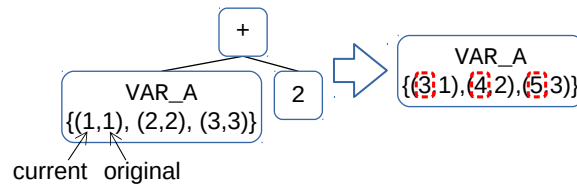
**Arithmetic Operator on two Literals** For integer arithmetic operations on two literal values, the result is simply calculated.



In this figure, the expression  $2 * 4$  results in the literal value 8.

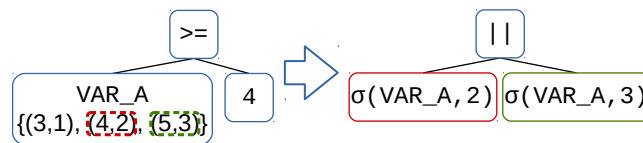
**Comparison Operator on two Literals** For comparison operations on two literal values, the resulting Boolean constant is simply calculated. Neither side of the comparison contains any integer variables, thus a single Boolean literal can express the satisfiability of this (sub-)expression.

**Arithmetic Operator on Literal and Variable** For integer arithmetic operations on variables, the operation is calculated on each of the allowed values. A set of tuples is stored in the variable, which contains for each *original* value, the *currently* computed value. All arithmetic operations on this variable will always update the *current* value. When resolving the variable to a Boolean formula later on, it is important which *original* value led to the currently computed one.



In this figure, the literal value 2 is added to  $\text{VAR\_A}$ .  $\text{VAR\_A}$  has three possible values: 1, 2, and 3. For each of these possible values, the operation is computed and the result stored in the first component in the tuple. The second component is not modified; it contains the *original* value of  $\text{VAR\_A}$  that led to the *currently* computed one. For example, in the second tuple of the result, the *original* value 2 of  $\text{VAR\_A}$  led to the *current* value of 4 (via the addition of 2).

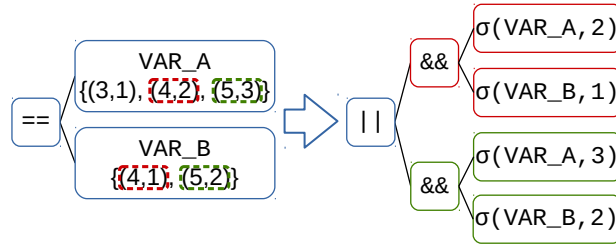
**Comparison Operator on Variable and Literal** A comparison of a variable and an integer literal is resolved to a propositional formula, which contains all possible *original* values that satisfy the comparison. The comparison is computed on all the *current* values stored in the variable. For each *current* value that satisfies the comparison, the corresponding *original* value of the variable is turned into a Boolean variable via the  $\sigma$  function. All these Boolean variables that fulfill the comparison are then combined with a Boolean disjunction operator.



In this figure,  $\text{VAR\_A}$  is compared with the literal 4 with a “greater than or equal” comparison operator. In this example, the *current* and *original* values in the tuples

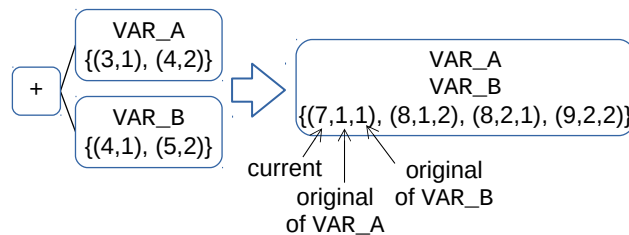
are different; this is because some previous arithmetic operation on `VAR_A` has modified them. This is not always the case (a variable can also be compared without doing arithmetic on it first), but we chose this for illustration purposes, to make it clear that the *current* and *original* values have to be treated differently. Two of the *current* values (the first component of the tuples) of `VAR_A` fulfill the comparison: the second and the third tuple. From both these tuples, the *original* values (the second component) are transformed into Boolean variables and combined with a logical disjunction.

**Comparison Operator on two Variables** A comparison of two variables is resolved to a propositional formula that contains all possible combinations of *original* values that satisfy the comparison. For each pair of the *current* values of the two variables it is checked if they fulfill the comparison operator. For each pair that does fulfill it, the two *original* values of the variables are turned into Boolean variables (via the  $\sigma$  function) and combined with a logical conjunction operator. All of these conjunction terms are then combined with a logical disjunction operator.



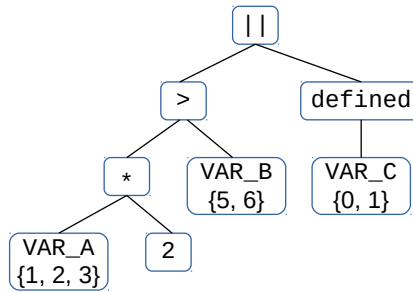
In this figure, there are two pairs of tuples that have the same *current* value and thus fulfill the equality operator: the second one from `VAR_A` and the first one of `VAR_B` both have the value 4, the third one from `VAR_A` and the second one from `VAR_B` both have the value 5. For the first pair, the *original* value of `VAR_A` that led to the *current* value is 2 (the second component in the tuple), while the *original* value of `VAR_B` is 1. Thus, the Boolean representation of this combination is  $\sigma(\text{VAR\_A}, 2) \wedge \sigma(\text{VAR\_B}, 1)$ . Similarly, the Boolean representation of the second matching pair is  $\sigma(\text{VAR\_A}, 3) \wedge \sigma(\text{VAR\_B}, 2)$ . Since both of these pairs fulfill the equality operator, they are combined with a disjunction operator.

**Arithmetic Operator on two Variables** For integer arithmetic operations on two variables, the operation is done on each combination of the *current* values of both variables. For each of these calculated values, both of the *original* values of the variables that led to this *current* value are stored. When turning this tuple into a Boolean formula, instead of a single variable being created (e.g.  $\sigma(\text{VAR\_A}, 2)$ ), a logical conjunction of the two variables is created based on the *original* values stored in the tuple (e.g.  $\sigma(\text{VAR\_A}, 2) \wedge \sigma(\text{VAR\_B}, 1)$ ).

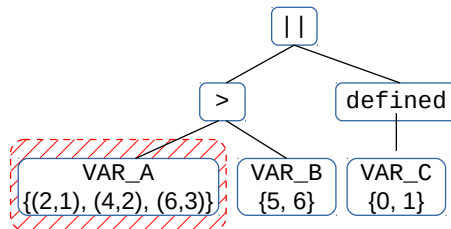


In this figure, the two variables `VAR_A` and `VAR_B`, both with two possible values, are added together. Combining the first tuple of both, results in the first tuple of the result: the *current* values (the first components of the tuples: 3 and 4) are added together, resulting in the new *current* value 7. Then, both of the *original* values (the second components in the tuples) are stored in the result, to indicate which *original* values of `VAR_A` and `VAR_B` led to the current value of 7. When turning this tuple into a Boolean formula (if the *current* value of this tuple fulfills a comparison later on), then the *original* values of both variables have to be considered: the resulting formula is  $\sigma(\text{VAR\_A}, 1) \wedge \sigma(\text{VAR\_B}, 1)$ . There are two different combinations that lead to the *current* value of 8. Thus, the result has two tuples where the first component is 8, but with different *original* values in the following components.

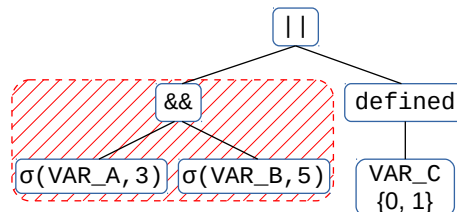
The following example illustrates how these transformation rules are applied. The AST of the condition in Formula 3.1, shown in the following figure, is used as a basis. The allowed values for each variability variable has been annotated.



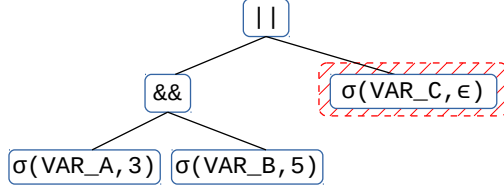
First, the multiplication operator is resolved by multiplying each possible value of `VAR_A` by the literal value 2. The highlighted region shows what part of the AST has been changed in this step.



Now, the comparison operator can be resolved. The *current* values of `VAR_A` and `VAR_B` are compared. Since there was no arithmetic operation on `VAR_B`, its *original* and *current* values are identical. Only one pair of *current* values of `VAR_A` and `VAR_B` match: the third one of `VAR_A` is greater than the first one of `VAR_B` ( $6 > 5$ ). This is then converted into a propositional formula, that specifies that the original value 3 from `VAR_A` and 5 from `VAR_B` fulfill this comparison.



Finally, the `defined(VAR_C)` call is replaced by the  $\epsilon$  variable for `VAR_C`.



The AST now only contains Boolean variables and operators. The final step is to convert it back into a C-preprocessor condition string. This involves applying the  $\sigma$  function to get the names of the Boolean variables and adding a `defined` call around them. This results in the this condition:

```
#if (defined(VAR_A_eq_3) && defined(VAR_B_eq_5)) || defined(VAR_C)
```

In many cases, the resulting propositional formula is much larger, compared to the original integer-based one. The goal is not to produce small or readable conditions, but to provide input data for analysis tools. Thus, it is not a primary concern to keep the resulting propositional formulas concise. However, in practice, we encountered runtime and memory problems with too large conditions in a very small number of cases.

To circumvent the runtime and memory problems of too large conditions, our tool defines a fixed upper limit for the number of value combinations to consider. When a series of arithmetic operations on variables exceeds this limit, we drop the per-value analysis for this sub-expression. Instead, we fall back to a less exact approach: only  $\sigma(var, \epsilon)$  variables are created for all involved variables. This retains the information, that the condition *somehow* depends on these variables, but the information which concrete values it depends on is lost.

A special case in the transformation process are integer variables that have only one allowed value. In the PS-EC product line, the legacy variability variables fall into this category. In the past, these were variability variables, but now their variability has been dropped and they are defined to be constants. If these variables are encountered in the non-Boolean transformation process, they are replaced by their constant value, effectively turning them into integer literals. This makes it easier to calculate the combinations of the other integer variability variables, that satisfy a constraint. It also removes unnecessary variables from the propositional formulas, and thus reduces the unnecessary complexity of the output.

Another special case are integer variables that have no restriction on the allowed values. The “allowed values” for such a variable are for example the whole range of 32 bit integer variables. It is not possible (or at least not feasible) to introduce Boolean variables for each of the possible values. In this case, a less exact approach is applied, to still be able to handle these variables: only the single Boolean variable  $\sigma(var, \epsilon)$  is introduced. Then, in each condition where the unrestricted variable appears, this Boolean variable is used, no matter which actual value of the variable would fulfill the condition. This way, the variability information that the given condition *somehow* depends on the unrestricted variable is preserved; however, the information which specific values it depends on is lost. For example, the condition `VAR_D + 2 > 5`, with `VAR_D` as an unrestricted integer variable, is converted to  $\sigma(\text{VAR\_D}, \epsilon)$ .

This inexact strategy has few drawbacks. The resulting formulas for unrestricted integer variables will only represent that the original condition somehow makes use of the unrestricted variable. This means that the final analysis steps, that analyze the variability conditions, only have vague information for these kind of variables. Additionally, the inexact approach may also result in wrong representations of the satisfiability of variability conditions. For instance, the unsatisfiable condition  $\text{VAR\_D} > 0 \wedge \text{VAR\_D} < 0$  would be transformed to  $\sigma(\text{VAR\_D}, \epsilon) \wedge \sigma(\text{VAR\_D}, \epsilon)$ , which appears satisfiable.

The impact of the inexact strategy for unrestricted variables depends on the concrete use-case where this approach is applied. The more unrestricted integer variables are present in the variability model, the more of a problem this becomes. Also the usage of these variables in the conditions needs to be examined: if the unrestricted variables are mixed together with the restricted integer variables in the conditions, then the inexact results may influence the other variables, too. In contrast, if they are mostly used in separate conditions, then the results for the restricted variables are unaffected and remain exact.

In addition to the potential problems mentioned above, there are a few minor technical issues with the implementation in KernelHaven. These stem from the specifics of the C-preprocessor, and are thus not inherent to the approach itself.

- The C-preprocessor has no well-defined data types. This leads to a problem when evaluating the bit-wise negation operator  $\sim$ , where the concrete type of an integer (bit size, and whether it is signed or unsigned) is important for correct results. However, in practice, this operator is not used much.
- Based on the usage of variability in the PS-EC product line, the implementation is only designed to handle integer and Boolean variables in the C-preprocessor conditions. It can not handle string variables, or the string concatenation operator (`##`). When encountering this, the tool will print a warning and skip replacing the condition.
- The C-preprocessor allows defining functions (with the `#define` directive) that can be used in `#if`-statements. The implementation in KernelHaven can not evaluate these custom functions; when such a function appears in a condition, the tool will print a warning and skip replacing the condition.

### 3.3 Extraction of C-preprocessor Conditions

This section describes how the CodeBlockExtractor extracts variability conditions from the C-preprocessor. The CodeBlockExtractor is a code extractor plug-in for KernelHaven [Kra18], that has been developed in the context of this thesis. It considers Boolean variability, as implemented in the Linux Kernel [Lin19] and other software product lines using the C-preprocessor. Section 3.2 describes how the integer-based variability of the Bosch PS-EC product line is converted to pure Boolean variability as handled in this section.

The C-preprocessor contains statements for conditional compilation: `#if`, `#elif`, `#else`, `#endif`, `#ifdef`, and `#ifndef`. These control whether the lines that are enclosed by them are passed to the compiler, or whether they are ignored during compilation. This is used to implement variability in the source code: variability variables are used in the conditional compilation expressions, that control which variant of the source code should be compiled.

```

1 statement1;
2 #if defined(VAR_A) || !defined(VAR_B)
3     statement2;
4     #if defined(VAR_C)
5         statement3;
6     #elif defined(VAR_B) && (defined(VAR_D) || !defined(VAR_E))
7         statement4;
8     #else
9         statement5;
10    #endif
11 #else
12     statement6;
13     #ifdef VAR_C
14         statement7;
15     #endif
16 #endif

```

Figure 3.5: Conditional compilation blocks using the C-preprocessor

Figure 3.5 shows an example of C-preprocessor blocks. Five Boolean variability variables control which statements are compiled: `VAR_A` through `VAR_E`. Their two possible states, true and false, are encoded by either defining or not defining them as C-preprocessor symbols; thus, the `defined` function of the C-preprocessor is used to check whether the variability variables are true or false. `statement1` in line 1 is outside of any C-preprocessor blocks; it is always included in the compilation process. In contrast, `statement2` in line 3 is nested inside an `#if` block; it is only included if the expression of that `#if` block evaluates to true. In this case, `VAR_A` needs to be defined or `VAR_B` needs to be undefined.

The goal of the `CodeBlockExtractor` is to extract the variability information that is present in the C-preprocessor blocks and to represent it in a data structure that will be used by a following analysis. This data structure must represent which parts of the source code are variable and what the presence conditions of these parts are. The presence condition of a source code block is a Boolean formula that, using variability variables, describes under which conditions that block is included in the product (i.e. under which conditions it is compiled).

The `CodeBlockExtractor` walks line-by-line through the source code file. C-preprocessor statements are lines starting with a hash character (`#`); optionally, whitespace may be before or after the hash. Lines inside block comments (`/* ... */`) are not considered by the C-preprocessor. When a C-preprocessor statement that opens a new conditional compilation block (`#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`) is encountered, a `CodeBlock` object is created by the `CodeBlockExtractor` with the starting line set to the opening line number. When a statement that closes a conditional compilation block (`#endif`, `#elif`, `#else`) is encountered, the `CodeBlockExtractor` sets the end line for the previously created `CodeBlock` object. The `#elif` and `#else` statements both close the previous block and directly open a new one.

Conditional compilation blocks may be nested inside each other. For instance, the `#if` in line 4 in Figure 3.5 is nested inside the `#if` in line 2. The `CodeBlockExtractor` keeps track of this by storing the currently open `CodeBlock` objects in a stack. A newly opened `CodeBlock` is pushed onto the stack, a closed `CodeBlock` pops the top of the stack. Additionally, when a new `CodeBlock` is created, it is added to the list of children of the enclosing `CodeBlock`

(i.e. the `CodeBlock` that is at the top of the stack). Blocks that are not nested inside other blocks are called *top blocks* of the source code file.

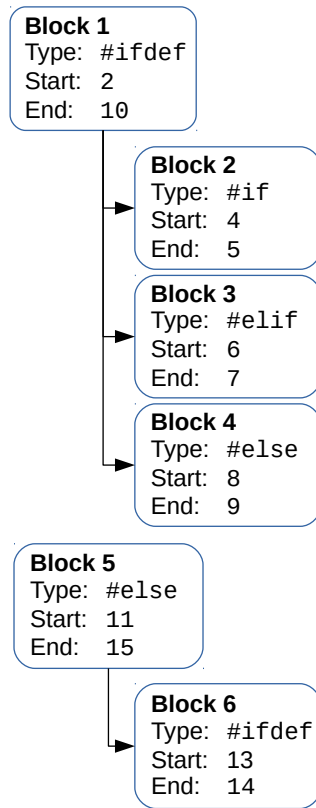


Figure 3.6: Nesting structure of `CodeBlocks`

Figure 3.6 illustrates the nesting structure of the blocks in Figure 3.5. Block 1 and Block 5 are top blocks, as they are not nested in any other blocks. The arrows indicate the children of blocks. Blocks 2, 3, and 4 are nested inside the `#if` in line 2 (Block 1); Block 6 is nested inside the `#else` in line 11 (Block 5).

The `CodeBlockExtractor` also considers whether there is code outside of all conditional compilation blocks. If this is the case, an artificial `CodeBlock` is created that covers the whole file. For instance, `statement1` in line 1 of Figure 3.5 is outside of all C-preprocessor blocks. Thus, the `CodeBlockExtractor` will create a `CodeBlock` with the presence condition `true` that covers the whole file (lines 1 through 16). The previous top blocks of the file, Block 1 in line 2 and Block 5 in line 11 will be nested as children in this new artificial block. This structure allows the following analysis steps to differentiate whether all code is inside conditional compilation blocks, or whether there is also common code in the source file.

The presence condition of a conditional compilation block depends on its *expression*, its *type*, the *previous siblings*, and the *enclosing block* [STL+10]. The *expression* of a block  $b$ ,  $expr(b)$ , is the parsed Boolean variability condition that is written in the source code. The `CodeBlockExtractor` parses the C-preprocessor expression to a Boolean formula. For instance, the `#if` statement in line 2 of Figure 3.5 (*block1*) has  $expr(block1) = \text{VAR\_A} \vee \neg \text{VAR\_B}$ . The `#else` statement in line 11 (*block5*) does not have an expression; in this case,  $expr(block5)$  is defined to be *true*.

The C-preprocessor supports expressions that contain more than pure Boolean logic. For instance, arithmetic and comparison operators for numbers are supported. However, the CodeBlockExtractor is only meant to support Boolean variability expressions. Only the Boolean operators (`&&`, `||`, `!`) and Boolean variables (where the `defined` function is used to differentiate between true and false) are supported.

The expression of a conditional compilation block in the source code is parsed with full support for the C-preprocessor expression syntax. After that, if unsupported operations (e.g. number comparisons or unknown macro calls) are found in the resulting abstract syntax tree, an exception is thrown. The user can configure what should happen in this case: parsing of the whole file can be aborted, or parsing of the expression can be aborted and a replacement is used instead. In the latter case, it can be configured to either use the constant `true` as a replacement, or a special variable called `PARSING_ERROR`. The `true` constant will likely cause the condition of this block to disappear in further analysis, as constants are usually pruned from Boolean formulas. In contrast, the `PARSING_ERROR` variable will be visible in the analysis results, and signal which parts of the analysis result are affected by parsing errors.

The CodeBlockExtractor can also be configured to parse in a mode called *fuzzy parsing*. In this mode, some operations on numeric values are allowed and encoded as special Boolean variables. Comparisons between numbers and variables and comparisons between two variables are supported. For each such comparison, a Boolean variable is created to represent it. This follows the underlying idea of the  $\sigma$  function as introduced in Section 3.2. For example, if the comparison `VAR_A > 6` is encountered, a Boolean variable `VAR_A_gt_6` is used to replace the comparison. This *fuzzy parsing* approach makes the parser more accepting, which may improve the quality of the created variability information. However, it also introduces new artificial variables, that need to be considered in the following analysis. With a non-Boolean transformation of the variability conditions beforehand, as presented in Section 3.2, there is no need to enable *fuzzy parsing*, as all numeric operations have already been resolved.

The variability implementation of the Linux Kernel served as the basis for the kind of C-preprocessor variability the CodeBlockExtractor supports. Boolean variability variables are realized by either defining or not defining a C-preprocessor symbol. The `defined` function is used in `#if` conditions to read the state of a variability variable. However, the Linux Kernel uses three custom macro functions in addition to the `defined` function: `IS_MODULE`, `IS_ENABLED`, and `IS_BUILTIN`. These are used to handle tristate variability variables, which are implemented using two Boolean C-preprocessor symbols. The CodeBlockExtractor has a setting which enables support for these three macros. The macro calls are replaced by Boolean formulas that use the two Boolean variables that make up the tristate variable. This way, the CodeBlockExtractor fully supports the Boolean and tristate variability of the Linux Kernel, which makes up most of the variability in the product line [BSL+13].

The presence condition of a conditional compilation block does not only depend on its expression, though. The *condition* of a block  $b$ ,  $cond(b)$ , describes how the *type* of C-preprocessor statement that is used influences the inclusion of the block. This is relevant for `#elif` and `#else` conditions: they are only included, if none of the previous siblings in that if-elif-else group are included. Let  $ps(b)$  be the set of all preceding siblings of block  $b$ . The condition of  $b$  is defined as:

$$cond(b) = \begin{cases} \bigwedge_{s \in ps(b)} \neg expr(s) & \text{if } b \text{ is an } \#else \\ expr(b) \wedge \bigwedge_{s \in ps(b)} \neg expr(s) & \text{if } b \text{ is an } \#elif \\ expr(b) & \text{else} \end{cases}$$

For example, the `#elif` block in line 6 of Figure 3.5 (*block3*) is included, if its expression evaluates to *true*, and the expression of the preceding `#if` in line 4 (*block2*) does *not* evaluate to *true*:

$$cond(block3) = \underbrace{VAR\_B \wedge (VAR\_D \vee \neg VAR\_E)}_{expr(block3)} \wedge \underbrace{\neg VAR\_C}_{\neg expr(block2)}$$

The `#else` block (*block4*) in line 8 is included, if both, the expression of the `#if` in line 4 and the expression of the `#elif` in line 6 do *not* evaluate to *true*:

$$cond(block4) = \underbrace{\neg VAR\_C}_{\neg expr(block2)} \wedge \underbrace{\neg (VAR\_B \wedge (VAR\_D \vee \neg VAR\_E))}_{\neg expr(block3)}$$

Finally, the nesting structure of the conditional compilation blocks needs to be considered to calculate the presence condition. A block *b* is only included, if its parent block *pb* is also included. The presence condition *pc(b)* is defined as:

$$pc(b) = cond(b) \wedge pc(pb)$$

If *b* is a top block, i.e. there is no enclosing block, *pc(pb)* can be considered to be *true*. For example, the presence condition of the `#if` block in line 2 in Figure 3.5 (*block1*) is only its condition (*pc(block1) = cond(block1)*), as it does not have any enclosing block. In contrast, the `#elif` block in line 6 (*block3*) is a child of *block1*; its presence condition is:

$$pc(block3) = \underbrace{VAR\_B \wedge (VAR\_D \vee \neg VAR\_E) \wedge \neg VAR\_C}_{cond(block3)} \wedge \underbrace{(VAR\_A \vee \neg VAR\_B)}_{pc(block1)}$$

### 3.4 Feature-Effect Analysis

The goal of the analysis in this chapter is to create a set of constraints for the configuration process of variability variables. These constraints encode domain knowledge that defines which configurations of the variables are not allowed. The constraints can be used in the configuration process to automatically exclude invalid options, which has the potential to reduce the effort and increase the accuracy of creating a new configuration (cf. Section 3.1).

```

1 // Block b1  pc(b1) = VAR_A
2 #ifdef VAR_A
3     statement1;
4     // Block b2  pc(b2) = VAR_A && VAR_B
5     #ifdef VAR_B
6         statement2;
7     #endif
8 #endif

```

Figure 3.7: C-preprocessor conditions for illustrating feature-effects

The idea of the analysis in this chapter is to extract domain knowledge that is implicitly present in the variability conditions of the product line. This can be used to create a set initial constraints that can be built upon by domain experts in the future.

The feature-effect<sup>3</sup> analysis approach [NBK+14] is used to extract the constraints from the variability conditions of the product line. The basic assumption of this approach is that toggling a variability variable should always have an effect on the final product. For instance, the two C-preprocessor blocks in Figure 3.7 control the inclusion of one statement each. Toggling `VAR_A` always toggles the inclusion of `statement1` in line 3; thus `VAR_A` always changes the resulting product. However, `VAR_B` only controls the inclusion of `statement2`, if `VAR_A` is set. If `VAR_A` is not set, then the whole block from line 2 to 8 is not included, no matter how `VAR_B` is set. Together with the assumption, that toggling a variability variable is meant to have an effect on the product, it can be concluded that `VAR_B` is not meant to be selected when `VAR_A` is deselected. This relation can be encoded as the constraint:  $\text{VAR\_B} \Rightarrow \text{VAR\_A}$ . With this constraint in the variability model, it can be assured that all valid mutations to the configuration will have an effect on the created product.

More generally, an effect on the product is assumed when the inclusion of any conditional compilation block changes. A more detailed analysis whether the de-/selection of a block actually results in a semantically different product is undecidable, thus this approximation is chosen. For a single block  $b$  with the presence condition  $pc(b)$ , the effect of the variable  $v$  on the block can be expressed by the formula:

$$pc(b)[v \leftarrow true] \oplus pc(b)[v \leftarrow false]$$

where  $F[v \leftarrow x]$  means setting all occurrences of  $v$  in formula  $F$  to  $x$  and  $\oplus$  denotes the XOR operation. The variable  $v$  is set to each, true and false, and the XOR checks whether the two resulting formulas differ. As an example, consider the two blocks  $b1$  and  $b2$  in Figure 3.7. The effect of `VAR_A` on block  $b1$  is:

$$\begin{aligned}
 & pc(b1)[\text{VAR\_A} \leftarrow true] \oplus pc(b1)[\text{VAR\_A} \leftarrow false] \\
 &= true \oplus false \\
 &= true
 \end{aligned}$$

<sup>3</sup>The authors of the feature-effect approach use the term “feature”, which implies a user-facing capability of the software product. In this thesis, we use the term “variability variable” to include, more generally, all variables that are used to implement variability in a software product line. For the sake of this analysis, the terms “feature” as used by Nadi et. al. and “variability variable” can be seen as synonymous.

This means, that `VAR_A` always has an effect on `b1`, without any precondition. In contrast, the effect of `VAR_B` on `b2` is:

$$\begin{aligned}
 & pc(b2)[\text{VAR\_B} \leftarrow \text{true}] \oplus pc(b2)[\text{VAR\_B} \leftarrow \text{false}] \\
 &= (\text{VAR\_A} \wedge \text{true}) \oplus (\text{VAR\_A} \wedge \text{false}) \\
 &= \text{VAR\_A} \oplus \text{false} \\
 &= \text{VAR\_A}
 \end{aligned}$$

As shown earlier, the effect of `VAR_B` on this block depends on `VAR_A` being set to true.

The effect of a variable on a given block can also result in more complex formulas. For example, for a block  $b$  with  $pc(b) = \text{VAR\_A} \vee (\text{VAR\_B} \wedge \text{VAR\_C})$ , the variable `VAR_A` has the effect:

$$\begin{aligned}
 & pc(b)[\text{VAR\_A} \leftarrow \text{true}] \oplus pc(b)[\text{VAR\_A} \leftarrow \text{false}] \\
 &= (\text{true} \vee (\text{VAR\_B} \wedge \text{VAR\_C})) \oplus (\text{false} \vee (\text{VAR\_B} \wedge \text{VAR\_C})) \\
 &= (\text{true}) \oplus (\text{VAR\_B} \wedge \text{VAR\_C}) \\
 &= \neg(\text{VAR\_B} \wedge \text{VAR\_C}) \\
 &= \neg\text{VAR\_B} \vee \neg\text{VAR\_C}
 \end{aligned}$$

The feature-effect analysis should create constraints for the whole product line, not just individual blocks. A variability variable has an effect on the product, if *any* conditional compilation block is affected by it. Thus, all conditional compilation blocks of the product line have to be considered. Let  $B$  be the set of all conditional compilation blocks of the product line; the feature-effect  $fe(v)$  of the variability variable  $v$  is:

$$fe(v) = \bigvee_{b \in B} pc(b)[v \leftarrow \text{true}] \oplus pc(b)[v \leftarrow \text{false}] \quad (3.2)$$

For the variables in Figure 3.7 this means:

$$\begin{aligned}
 fe(\text{VAR\_A}) &= (pc(b1)[\text{VAR\_A} \leftarrow \text{true}] \oplus pc(b1)[\text{VAR\_A} \leftarrow \text{false}]) \\
 &\quad \vee (pc(b2)[\text{VAR\_A} \leftarrow \text{true}] \oplus pc(b2)[\text{VAR\_A} \leftarrow \text{false}]) \\
 &= \text{true} \vee \text{VAR\_B} \\
 &= \text{true} \\
 fe(\text{VAR\_B}) &= (pc(b1)[\text{VAR\_B} \leftarrow \text{true}] \oplus pc(b1)[\text{VAR\_B} \leftarrow \text{false}]) \\
 &\quad \vee (pc(b2)[\text{VAR\_B} \leftarrow \text{true}] \oplus pc(b2)[\text{VAR\_B} \leftarrow \text{false}]) \\
 &= \text{false} \vee \text{VAR\_A} \\
 &= \text{VAR\_A}
 \end{aligned}$$

The disjunction over the effects of a variability variable on individual blocks expresses that the variable should have an effect on *any* block in the product line. This mechanism leads to a few interesting observations regarding the resulting feature-effect:

- Presence conditions of blocks that do not contain the variable that the feature-effect is computed for, do not influence the resulting feature-effect formula. Setting the variable to true and false leaves the presence conditions of these blocks unchanged, thus the XOR over the two unchanged formulas results in the constant *false*. These *false* constants are then removed when building the disjunction over the individual effects (they cancel out as  $x \vee \text{false} = x$ ).

- The feature-effect can only become *false*, if all parts of the disjunction are *false*. That would mean, that the variable never has any effect on any conditional compilation block of the product line.
- The disjunction over the individual block-effects causes the more relaxed terms to cancel out the more restrictive ones. For instance, a variable that appears in two blocks may have the individual effects  $A$  and  $A \wedge B$ . That means, for one block it only depends on  $A$ , while for the other it depends on  $A$  and  $B$ . When combining both effects with a disjunction, the more relaxed dependency on only  $A$  cancels out the more restrictive dependency on  $A$  and  $B$  ( $A \vee (A \wedge B) = A$ ). The resulting feature-effect contains only  $A$ .
- A variable that appears at least once as a top-level variable, that is it appears in a block where it always has an effect, the feature-effect becomes *true*. For instance, `VAR_A` in Figure 3.7 always has an effect on block *b1* (the individual block-effect is *true*), but only has an effect on block *b2* if `VAR_B` is set. When combining both, the *true* formula for block *b1* causes the whole disjunction, and thus the whole feature-effect, to become *true* as well ( $x \vee \text{true} = \text{true}$ ).
- If a variable is always nested inside another one (that is, the parent variable appears as a conjunction in all presence conditions), then the parent variable appears with a conjunction in the feature-effect ( $(p \wedge x) \vee (p \wedge y) = p \wedge (x \vee y)$ ).
- If a variable has two parents  $p1$  and  $p2$  (that is, the variable is always either nested in  $p1$  or  $p2$ ), then the feature-effect becomes  $p1 \vee p2$ .

The implementation of the feature-effect analysis as an analysis plug-in for KernelHaven [KE19a] executes the following steps:

1. Collect a set of all unique presence conditions in the product line. Multiple occurrences of the same presence condition do not influence the result, thus only a single instance of each presence condition is required.
2. For each variability variable, create the subset of the presence conditions containing only the conditions where the variable appears. As noted above, if the variable does not appear in a presence condition, that presence condition does not influence its feature-effect.
3. For each variable with its set of presence conditions, the feature-effect formula (cf. Formula 3.2) is computed.

For the integer-based variability in the Bosch PS-EC product line, a few adaptations to the standard feature-effect computation are implemented. The non-Boolean transformation (cf. Section 3.2) creates artificial Boolean variables for each allowed value of the integer variability variables. The feature-effect analysis is executed on these Boolean variables. Thus, the resulting feature-effects are per allowed value of the original integer variability variables. For instance, for a variable  $A$  with three allowed values (1, 2, and 3), this analysis produces feature-effects for four Boolean variables:  $\sigma(A, 1)$ ,  $\sigma(A, 2)$ ,  $\sigma(A, 3)$ , and  $\sigma(A, \epsilon)$ . This gives fine-grained information on when setting  $A$  to a specific value has an effect

The non-Boolean transformation has an implicit constraint that the Boolean variables for the possible values of an integer variable are mutually exclusive. For instance,  $\sigma(A, 1)$  and  $\sigma(A, 2)$  may not be set to *true* at the same time. This constraint is not explicitly

modeled in the non-Boolean transformation process. Thus it may occur, that the feature-effect  $fe(\sigma(A, 1))$  contains the variable  $\sigma(A, 2)$ . For example,  $fe(\sigma(A, 1)) = x \vee \sigma(A, 2)$ ; i.e. setting  $A$  to 1 has an effect if  $x$  is set, or if  $A$  is set to 2. The latter case makes no sense, as  $A$  can not freely be set to 1 if it is already set to 2. The feature-effect computation is adapted to prevent this problem. For an integer variability variable  $v$ , with  $R(v)$  denoting the set of allowed integer values of  $v$ , the following replacement is done in the computed feature-effect:

$$\forall i, j \in R(v), i \neq j \mid fe(\sigma(v, i))[\sigma(v, j) \leftarrow false]$$

In the feature-effect of the Boolean variable  $\sigma(v, i)$  (i.e.  $v$  being set to a value  $i$ ), all variables that denote that  $v$  is set to another value  $j$  are replaced by *false*. In the example above, this means that in  $fe(\sigma(A, 1)) = x \vee \sigma(A, 2)$  the variable  $\sigma(A, 2)$  is replaced by *false*:  $fe(\sigma(A, 1)) = x \vee false = x$ .

Another adaptation is required as the analysis of feature-effects per-value may be too fine-grained. The goal of the analysis is to create a set of basic constraints as a starting point for a variability model; a per-value analysis may result in too fine-grained formulas for that goal. Thus, an (optional) final step is implemented, that aggregates the feature-effect formulas of the artificial Boolean variables that were introduced for the integer variability variables. For an integer variable  $v$  and  $R(v)$  the function that returns the set of allowed integer values for  $v$ , the aggregated feature-effect is computed as:

$$fe(v) = \bigvee_{i \in R \cup \{\epsilon\}} fe(\sigma(v, i))$$

This disjunction over the feature-effects of the individual values has the same reasoning as the disjunction in the feature-effect formula itself (cf. Formula 3.2). An effect on the product is assumed when *any* change in the variability of the assets occurs. For an integer variability variable, this means that configuring it has an effect if *any* of the possible values results in a change.

Note that the feature-effect  $fe(\sigma(v, \epsilon))$ , and the aggregated feature-effect  $fe(v)$ , do not generally equal. This may seem counter-intuitive, as  $\sigma(v, \epsilon)$  denotes that  $v$  is set to *any* value (cf. Section 3.2), and  $fe(v)$  is also constructed in a way that denotes that *any* value of  $v$  has an effect. The disparity stems from the fact that the constraint, that  $\sigma(v, \epsilon)$  is *true* when  $v$  is set to any value, is not explicitly modeled in the non-Boolean transformation process.  $\sigma(v, \epsilon)$  is only used when an explicit **defined** call is encountered in the C-pre-processor conditions. Thus, in the collection of presence conditions that is passed to the feature-effect analysis,  $\sigma(v, \epsilon)$  does *not* generally denote that  $v$  is set to any value.

### 3.5 Result Simplification

The goal of the analysis in this chapter is to create formulas that serve as a basis for domain experts to create variability model constraints. Humans will have to read, understand, and adapt the formulas created here. Thus, the formulas have to be *understandable* by humans. The extracted domain knowledge is unusable for this goal if it is encoded in formulas that the domain experts cannot work with. Formulas that are too complex for human understanding may be useful for automatic processing, but the goal of this analysis is to serve as a basis for a variability model that humans can understand and modify.

The formulas created in the feature-effect analysis are in propositional logic containing only Boolean variables. They use two binary operators ( $\wedge$  and  $\vee$ ), one unary operator ( $\neg$ ), the literals *true* and *false*, and Boolean variables. With the basic structure being relatively simple and widely established, the main challenge for understandable formulas is their size. Thus, the goal of creating *understandable* formulas is to find, for each feature-effect, the smallest equal propositional formula. Here, “small” means few operators and variables.

The feature-effect analysis creates, without additional simplification, rather large formulas, even for relatively small presence condition inputs. For instance, the presence condition  $(A \vee B) \wedge C$  has the following feature-effect for  $A$  (cf. Formula 3.2):

$$((\text{true} \vee B) \wedge C) \oplus ((\text{false} \vee B) \wedge C)$$

The XOR operator,  $\oplus$ , is not present in KernelHavens representation of Boolean formulas, thus it needs to be expressed using the other operators:<sup>4</sup>

$$(((\text{true} \vee B) \wedge C) \vee ((\text{false} \vee B) \wedge C)) \wedge (\neg((\text{true} \vee B) \wedge C) \vee \neg((\text{false} \vee B) \wedge C))$$

Even for this relatively simple presence condition (3 variables with 2 operators), the feature-effect analysis creates a large formula (8 variables, 13 operators, 4 constants). This becomes even worse the more presence conditions the variable  $A$  is used in; for each such presence condition, the individual block-effect is combined with a disjunction. A first, almost trivial step to simplify the resulting feature-effects is to prune the constants by applying the annihilator and identity rules:<sup>5</sup>

$$(C \vee (B \wedge C)) \wedge (\neg C \vee \neg(B \wedge C))$$

This still results in a formula with 6 variables and 7 operators. Further simplification can reduce this even further:

$$C \wedge \neg B$$

This smallest possible representation only contains 2 variables and 2 operators. From a readability perspective, this is the most useful result. This example shows, that Boolean formula simplification is required when the resulting feature-effects are interpreted by humans. A post-processing component, that simplifies all feature-effects to their smallest possible equivalent, greatly increases the value of the feature-effect analysis for this use-case.

A problem is that existing work on Boolean formula minimization that we found is not applicable in the use-case of this chapter. The method developed by Quine and McCluskey finds the representation of a Boolean formula, that has the smallest number of operators [Qui52; Qui55; McC56]. However, since the underlying problem of finding the shortest possible Boolean equivalent is NP-complete, the exact Quine-McCluskey algorithm has exponential runtime with respect to the number of unique variables in the formula. It is generally not considered to be feasible for formulas with a larger number of unique variables. However, this commonly occurs when analyzing presence conditions of software product lines.

Another existing approach is the Espresso logic minimizer [BHH+82]. In contrast to the Quine-McCluskey algorithm, it is not an exact approach, but rather uses a heuristic to

<sup>4</sup> $X \oplus Y \iff (X \vee Y) \wedge (\neg X \vee \neg Y)$

<sup>5</sup> $X \vee \text{false} \iff X, X \wedge \text{true} \iff X, X \vee \text{true} \iff \text{true}, X \wedge \text{false} \iff \text{false}$

find shorter equivalent formulas. Because of this, its runtime does not grow exponentially with more unique variables. However, it is not applicable in the use-case of this chapter, as it requires the full truth-table of the Boolean formula to minimize as an input. Creating this truth-table is not feasible, as it grows exponentially in size with the number of unique variables in the formula. As with the Quine-McCluskey algorithm, the Espresso tool was also created in the context of the digital circuit community. There, Boolean simplification is applied to reduce the number of hardware logic components required to implement a circuit. Large numbers of unique variables, i.e. inputs in their case, seems to be not a common challenge in that context. In contrast, in the analysis of presence conditions of software product lines, large amounts of different variables in a single formula regularly occur.

Propositional formulas can be represented as binary decision diagrams (BDDs) and the simplification can be conducted on such BDDs [Bry86]. A BDD is a rooted, directed, acyclic graph. A variable is represented as a node in the BDD, with two outgoing edges that represent the variable being set to *true* and *false*. A BDD is acyclic, so paths through the BDD represent specific configurations for the variables. All paths end in either of two terminal nodes, which represent the result of the propositional formula for that configuration of variables (*true* or *false*). Figure 3.8 shows a BDD representation of the formula  $A \vee (B \wedge \neg C)$ . The continuous edges represent that the variable is set to *true*, the dashed edges represent that the variable is set to *false*. The path  $\neg A, \neg B$  ends in the terminal node *false*, which means that the original is false if *A* and *B* are set to *false*. The path  $\neg A, B, \neg C$  ends in the terminal *true*; the original formula results in *true* for this configuration.

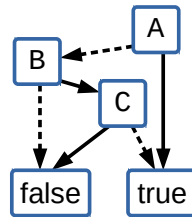


Figure 3.8: Example Binary Decision Diagram

BDDs are often used as a representation of propositional formulas to reduce their size. However, this approach is not feasible in the use-case of this chapter. This is because the construction of the BDD representation for large formulas is costly, as it requires setting variables to *true* and *false* and evaluating the formula each time to build the respective sub-trees of a node in the BDD. This is similar to the calculation of the complete truth-table, which was already discussed to be unfeasible.

The solution that is implemented for the use-case of this chapter is to implement custom heuristic approaches that reduce the size of formulas in a reasonable runtime. The general idea is to apply standard rules of Boolean algebra to detect parts of the Boolean formula that can be converted to a more concise form. Apart from a list of standard Boolean algebra laws, the main driver was manual inspection of formulas that were created by the feature-effect analysis. From them, common patterns that could be further simplified were included in the heuristic implementation.

The following list introduces the different implementation approaches that were created in the context of this thesis (except the “simple Boolean algebra” approach, which has not

been developed in this thesis). They are ordered from simple to more sophisticated, and generally build upon the previous approaches that were developed beforehand:

**Prune Constants** This most simple approach prunes constants and variable repetitions (e.g.  $x \vee x$ ) from the formula. This was also applied as a first simplification in the introductory example of this section. The Boolean algebra rules implemented here include *identity*, *annihilator*, *idempotence*, and *double negation*. The advantage of this approach is that it is very fast; it requires only a single iteration, bottom-up through the Boolean formula abstract syntax tree (AST). However, a disadvantage is that it only implements very rudimentary simplifications; as seen in the introductory example of this section, further simplification approaches are almost always required. In KernelHaven, this approach is referred to as the **SIMPLE** simplifier and implemented in the `FormulaSimplifier` class.

**Simple Boolean Algebra** This approach applies more rules from Boolean algebra to simplify the formula. These rules include, among the ones implemented in the previous approach, *complementation* and *absorption*. In KernelHaven, this approach is referred to as the **VISITOR** simplifier and implemented in the `FormulaSimplificationVisitor` class.

The complementation rules ( $\neg A \wedge A \iff \text{false}$  and  $\neg A \vee A \iff \text{true}$ ) are only implemented for exact matches of  $A$ . This means, that if  $A$  itself is a sub-formula, the complementation rule is only applied if both versions of  $A$  have the exact same AST representation. For example, the commutativity of the  $\wedge$  and  $\vee$  operators are not considered in this check. This approach makes the equality checks to detect this simplification rule fast. However, it also decreases the number of possible simplifications that can be detected like this.

The absorption rules ( $A \wedge (A \vee B) \iff A$  and  $A \vee (A \wedge B) \iff A$ ) are only implemented for  $A$  being a variable or a negated variable. This also greatly reduces the overhead for equality checks to find this pattern in formulas to simplify.

**More Complex Boolean Algebra** This approach applies all of the above Boolean algebra rules, but has an improved detection approach. In KernelHaven, this approach is implemented in the `FormulaSimplificationVisitor2` class.

The detection mechanism of the previous approaches is limited in detecting patterns that can be simplified. They only check if simplification rules can be applied on the exact operator structure of the AST. For instance, they can only detect the idempotence rule ( $A \wedge A \iff A$ ) if it appears directly under an operator. They could detect it in the formula  $B \wedge (A \wedge A)$  and simplify it to  $B \wedge A$ , but they are unable to detect it like this:  $A \wedge (B \wedge A)$ . Because of the associativity of  $\wedge$ , both formulas are equal. However, the second one does not contain  $A \wedge A$  directly, thus the previous approaches cannot detect this.

The “more complex Boolean algebra” approach improves the detection mechanism by addressing this shortcoming. The operator structure is flattened, instead of keeping the original structure of the binary AST. For instance, for the formula  $A \wedge (B \wedge A)$ , all three terms of the conjunction ( $A$ ,  $B$ , and  $A$ ) are considered at once. Then, when checking for patterns that can be simplified, all combinations of terms are considered. This way, the idempotence can be detected in any nesting structure, even if it cannot be found when considering any operator in the AST by itself.

However, the flattened structure of terms leads to a significant increase of equality checks, as all combinations of terms need to be considered now. To alleviate the runtime penalty of this, equality checks are only done on variables. There are no equality checks on more complex sub-formulas. For instance, the idempotence rule is only detected when the same variable occurs in the list of terms; recurrence of the same sub-formula is not found. In this regard, this approach cannot simplify some cases that the previous approach could simplify. This disadvantage can be eliminated by combining the “more complex Boolean algebra” approach with the next one, the “equal sub-formula detection”.

In addition to all of the the above Boolean algebra rules, the “more complex Boolean algebra” approach also applies the *distributivity* rules to factor-out common variables. In contrast to the previous approaches, this is beneficial in this approach, as it can consider more terms to find common factors. This is possible, since the hierarchy of equal operators is flattened. The previous approaches only consider operators with two operands, whereas this approach, by flattening the operator structure, can consider many terms at once.

**Equal Sub-Formula Detection** This approach detects equal sub-formulas, which is used to improve the detection mechanism for simplification possibilities. It is used together with the previous approach. In KernelHaven, this approach is implemented in the `SubTreeSimplifier` class.

The previous approach detects simplification possibilities by checking if a rule pattern matches with the given formula. For instance, if the pattern  $\neg A \vee A$  is found (i.e. two terms in a disjunction are negatives of each other), then the whole disjunction is replaced by *true* according to the complementation rule. However, these patterns are only detected when the symbols ( $A$  in this case) are variables. If  $A$  were a sub-formula, e.g.  $B \wedge C$ , then the pattern would not be detected by the previous approach. However, the complementation rule could theoretically be used to simplify this disjunction.

The sub-formula detection approach allows the previous approach to detect such cases by replacing equal sub-formulas with variables. For example, the sub-formula  $B \wedge C$  is found more than once in the formula to simplify. In this case, this approach would replace  $B \wedge C$  with a temporary variable, run the previous simplification approach, and then replace the temporary variable with  $B \wedge C$  again. This allows the previous simplification to detect cases where, for example, the sub-formula  $B \wedge C$  (which is then replaced by a temporary variable) is a part of the complementation rule. The number of possible simplifications detected by the previous approach is greatly increased by this approach.

This approach first detects all sub-formulas that occur more than once. The detection is implemented by walking through the AST of the formula and collecting the (deep) hashes of each operator. If two operators have the same (deep) hash, an additional check is done that they are actually equal to avoid hash collisions. This algorithm finds all sub-trees in the AST and detects when an equal sub-tree occurs more than once. The (deep) hash of operators is constructed in a way that the commutativity of the  $\wedge$  and  $\vee$  operators does not matter, i.e. switched operands are still detected as an equal sub-formula.

After all recurring sub-formulas are found, they are replaced one-by-one with temporary variables. After each replacement, the formula is passed to the previous simpli-

fication approach, and the temporary variable in the simplified formula is replaced by the original sub-formula. The whole process of sub-formula detection is looped, until no further changes are detected when simplifying the formula.

**Combined Approach** This approach is a combination of the above simplification approaches. In KernelHaven, this is referred to as `ADAMS_AWESOME_SIMPLIFIER` and implemented in the `AdamsAwesomeSimplifier` class.

This approach was developed on a set of large formulas that were found in the feature-effect analysis of the use-case of this chapter. It re-uses the above approaches and applies them in an order that was found to produce good results with a reasonable runtime on the given test set. The approach has the following steps:

1. Prune constants (see first approach above)
2. Move negations inwards as much as possible (e.g. via  $\neg(A \vee B) \iff \neg A \wedge \neg B$ )
3. Apply the simple Boolean algebra approach from above
4. Apply the more complex Boolean algebra approach from above
5. Apply the equal sub-formula approach from above (which internally uses the more complex Boolean algebra approach)
6. Move negations outwards as much as possible (reverse idea to step 2)
7. Apply the equal sub-formula approach again

This whole process is applied in a loop that iterates until the length of the formula (measured in characters of the string representation) no longer decreases. The shortest formula that was found in all iterations is returned as the simplified result.

The general idea of each iteration is to apply the faster approaches first, as this reduces the formula size early, before applying the the more costly approaches. In the development process, the runtime and the formula size reduction of each step was monitored. This, for example, showed that moving the negations inwards or outwards before applying the equal sub-formula approach leads to better simplification results. Additionally, repeating the whole loop until no further reduction is detected leads to better results with only slightly increased runtime, compared to a fixed number of iterations.

The above approaches are general in the sense that any arbitrary Boolean formula can be input to them for simplification. This can be used as a post-processing step after the feature-effect formulas have been created. However, there is also potential for simplification *during* the feature-effect computation. This uses specifics in the feature-effect computation to employ special simplification techniques. Figure 3.9 illustrates the feature-effect computation for a single variable on an abstract level. On the left, the presence conditions where the variable occurs are supplied as the input. Each of them is used to create an XOR term (cf. Formula 3.2), which are then all combined with a disjunction. This produces the final feature-effect formula. In addition to this process, Figure 3.9 also shows where simplifications are done using orange boxes. This includes several simplification steps that use the general approaches described above, and a special simplifying disjunction.

One specialized simplification technique applied here is that the general simplification approach described above is applied early and throughout the computation. Applying simplification on the smaller parts that make up the final large formula has the potential

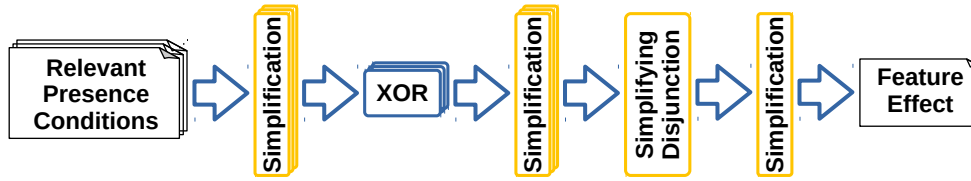


Figure 3.9: Simplification steps in the feature-effect computation

to decrease the required runtime. This is because simplification algorithms are generally faster on smaller formulas, since there are fewer possible combinations to consider when applying simplifications rules. The resulting simplified formulas then also lead to a smaller combined formula, which makes simplification on this faster. Additionally, the memory footprint also decreases, as the simplifications remove unnecessary sub-formulas as soon as possible, instead of carrying it throughout the computation process and removing it in the final simplification step.

In the feature-effect computation, this concept is applied by inserting simplification steps in-between each individual step. First, the presence conditions are simplified before any computation is done. Simplification here is usually cheap, as presence conditions are the smallest parts in the feature-effect computation. If simplification opportunities are found here, they carry throughout the remaining process and thus have a large impact. Then, after the XOR terms are created, they are also simplified. As shown in the introductory example of this section, the XOR terms contain constants which can always be simplified. Additionally, due to the structure of the XOR representation using  $\wedge$ ,  $\vee$ , and  $\neg$  operators, the individual formulas grow considerably in this process. Simplifying these formulas here, before they are combined to form an even larger formula, saves a lot of performance overhead for the final simplification. After the disjunction that combines all XOR terms, another final simplification step is applied. This then produces the final result.<sup>6</sup>

Another specialized simplification technique developed in the context of this thesis is implemented in the disjunction that combines the individual XOR terms. The idea here is to leave out disjunction terms that are already covered by the other terms in the disjunction. For instance, if the term  $A$  is already present in the disjunction, then adding the term  $A \wedge B$  next can be omitted. In this case, the first term is more general and covers the second, more restrictive, term completely. The other way around, if a newly added term is more general covering all existing terms in the disjunction, the existing terms can all be dropped in favor of the new term. For instance, the existing terms  $A \wedge B$  and  $A \wedge C$  can be dropped when  $A$  is added to the disjunction. In this case, only the more general term  $A$  is required to represent the whole disjunction. In the most extreme case, adding *true* to a disjunction completely overrides any previous or future terms added to it.

The underlying idea leveraged by this approach, that more general terms cancel out the more restrictive terms, goes along with the observations about feature-effects made in Section 3.4. For feature-effects, it is common that a few code blocks with a general block-specific feature-effect cause the more restrictive block-specific feature-effects of other code blocks to disappear. From a simplification perspective, where smaller results are more favorable, this is especially advantageous, as the restrictive terms that can be dropped are usually longer. For instance, the term  $A \wedge X$ , where  $X$  is a very large sub-formula, can be dropped if  $A$  is also a term in the disjunction.

---

<sup>6</sup>If the feature-effect formulas are aggregated (cf. Section 3.4), there is another simplification step after the aggregation.

The implementation of the simplifying disjunction uses SAT-solvers to determine which terms of the disjunction are more general than others. While the satisfiability problem is NP-complete, there is no runtime problem in practice. Modern solvers are very efficient, especially with the for SAT-solvers relatively small formulas, that occur in the feature-effect computation. Compared to the other simplification steps, the runtime of the SAT-solver calls is insignificant.

The simplifying disjunction keeps track of the previously added terms. For each newly added term, it is decided if all the previous terms or the newly added one are more general. A more exact approach would be to check for each combination of two terms, whether one is more general than the other. However, this is unfeasible, as it would result in a quadratically growing number of SAT-calls with respect to the number of terms. Instead, each newly added term (*new*) is compared to the disjunction of all previously added ones (*previous*). The first SAT-call is  $\text{sat}(\neg \text{previous} \wedge \text{new})$ , which checks if *new* is a subset of *previous*. If this SAT-call returns *false*, the *previous* terms are more general and cover the more restrictive *new* one; the *new* term is not added to the disjunction. The second SAT-call is  $\text{sat}(\text{previous} \wedge \neg \text{new})$ , which checks if *previous* is a subset of *new*. If this call returns *false*, *new* is more general and covers all terms in *previous*. In this case, all terms in *previous* are removed and only *new* is added. Finally, if both SAT-calls return *true*, then neither *new* nor *previous* are subsets of each other. In this case, the *previous* is kept and *new* is added to the disjunction; no simplification is done in this case.

### 3.6 Analysis Results

This section presents the results that were found in the application of the analysis presented in this chapter. This is a summary of the results found in the previously published work [EDK+18]. The analysis was conducted on 7 recent product variants which contain 120 % variability (cf. Section 3.1). The selection of the product variants aimed at creating a diverse product set, covering two different engine types: diesel (2 product variants) and gasoline (5 product variants). The feature-effect analysis conducted on these product variants created a result table for each of them. These tables list all the variability variables that are used in the respective product variant, and the feature-effects for these variables.

The feature-effects computed for the variability variables can be thought of as precondition that defines when configuring the variability variable has an effect on the final product (cf. Section 3.4). Variables with the feature-effect *true* always have an effect on the product; they are independent of all other variability variables. We call these *top-level* or *independent* variability variables, as these are on the top of the dependency hierarchy. In contrast, all other variability variables are *dependent* variables, as their effect on the product depends on other variables being configured a certain way.

The first analysis of the feature-effect results examines the proportion of dependent and independent variability variables in each of the analyzed product variants. Figure 3.10 depicts the found results. The absolute number of variability variables cannot be disclosed due to confidentially reasons. In this figure, the number of all variability variables in all product variants has been used as the 100 % baseline. The height of the bars signifies the relative number of variability variables used in each product variant. For example, 58.1 % of the variability variables encountered in all 7 product variants are present in the first product variant. Of these, about 68.7 % (i.e. 39.9 % of all variability variables)

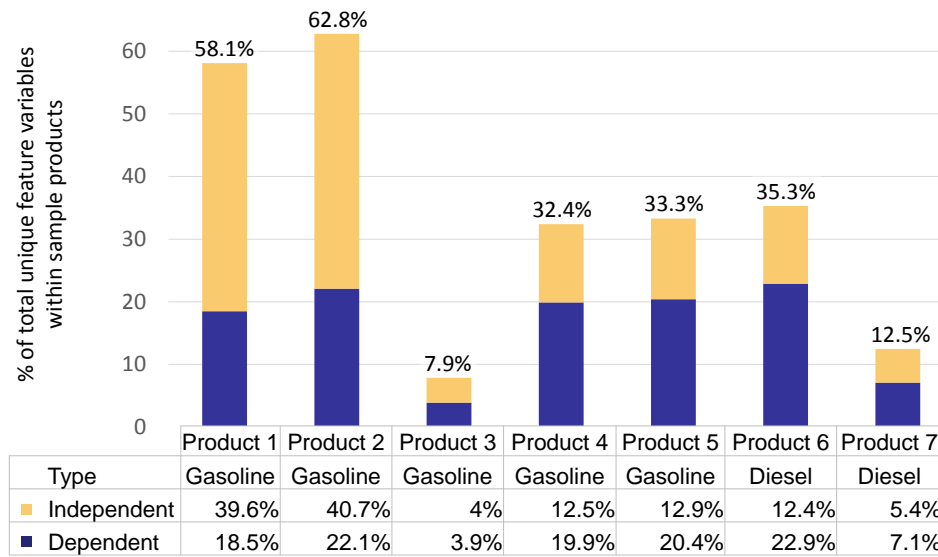


Figure 3.10: Ratio of dependent vs. independent variables per product variant [EDK+18]

are independent/top-level variables in this product variant. The total sum over of the number of used variability variables of the product variants is greater than 100 % as many variability variables occur in more than one product variant.

The next analysis of the feature-effect results aggregates the individual results of the 7 analyzed product variants. The aggregation compares the classifications into independent versus dependent variables in each product variant. If a variability variable is consistently dependent or independent it remains in that class. However, if a variability variable is dependent in some product variants, but independent in others, then it is classified as *mixed*.

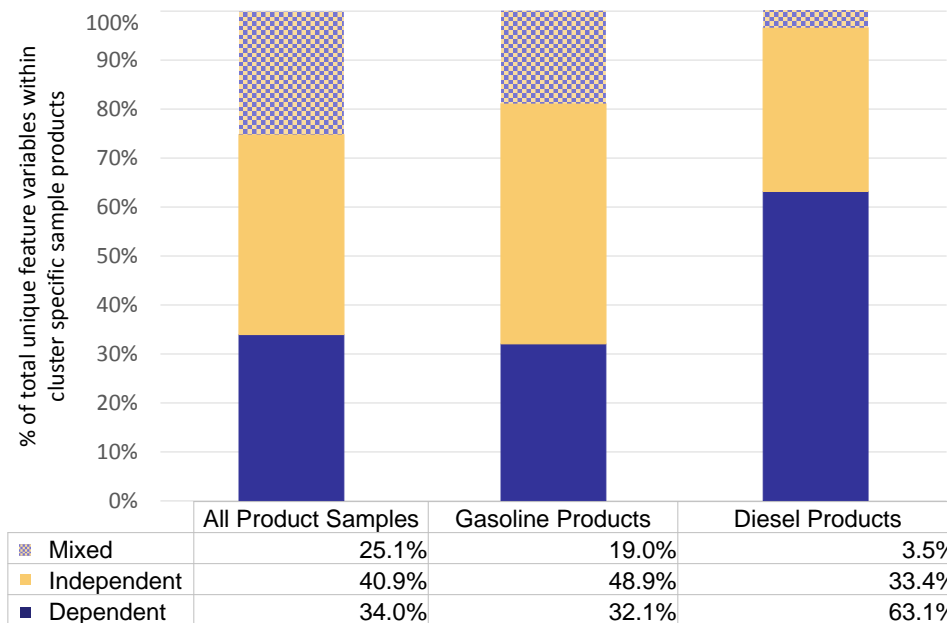


Figure 3.11: Aggregated classification of independent vs. dependent variables [EDK+18]

Figure 3.11 shows the results of this aggregated classification. The first bar depicts the ratio of the classifications when aggregating all product variants. The next two bars depict the ratio when only aggregating all gasoline (5 product variants which use 93.5 % of all variability variables) or diesel (2 product variants which use 37.9 % of all variability variables) product variants. This chart shows that aggregating more product variants leads to greater proportion of mixed variables, i.e. variables that are classified inconsistently across the individual product variants. For the specific product variants here, the high amount of mixed variables could be explained by their selection, which aimed at a high diversity of products. We would expect that a set of more similar products would lead to more consistent results regarding the classification of dependent and independent variables. This is already hinted in the chart, as the separate aggregation of the gasoline and diesel product variants results in more coherent classifications (i.e. less amount of mixed variables).

A final analysis of the feature-effect results aims at further exploring the relation of mixed variables and the number of product variants they appear in. We expect that the likelihood of a variability variable being classified as mixed increases with the number of product variants it appears in. This is simply because the classification of a variable can only change from consistent (i.e. consistently independent or consistently dependent) to mixed, and never from mixed to consistent. The statement, that a variable is consistently dependent or independent, increases in value when it holds over more product variants. For instance, a feature-effect that is found across many product variants likely represents more general domain knowledge than a feature-effect that is only found in a small number of product variants.

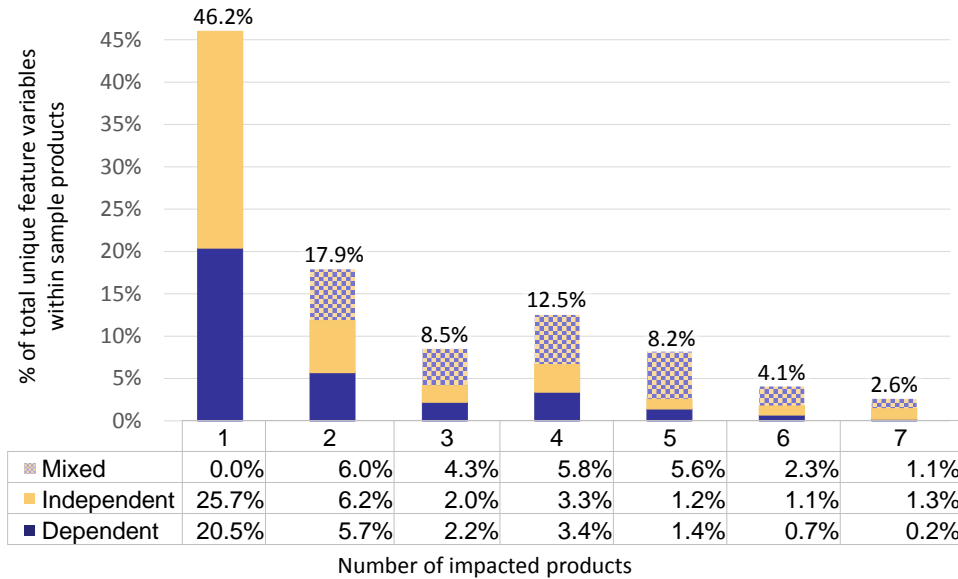


Figure 3.12: Classification of variables grouped by number of products [EDK+18]

Figure 3.12 depicts the classifications of variables into independent, dependent, and mixed grouped by the number of product variants they appear in. For instance, the first bar shows the variables that appear in exactly one product variant. In this case, no variable can be classified as mixed, as there can be no contradictory classifications in a single product variant. The second bar shows the classifications of the variables that appear in exactly two product variants, et cetera. The ratio of variables classified as mixed to non-mixed rises from 33.5 % for variables that appear in 2 product variants up to 68.3 % for variables appearing in 5 product variants; it then falls back down to 42.3 % for variables appearing

in all 7 product variants. This observation is somewhat consistent with the expectation that the ratio of mixed variables increases with the number of product variants they are found in. The overall amount of mixed variables (25.1 %) is greater than the amount of consistently dependent variables occurring in 2 or more product variants (13.6 %). This shows that the chance of finding a feature-effect that appears consistently across many product variants is quite low. This may stem from the fact that the analyzed product variants were selected to be diverse; a selection of more coherent product variants may have led to more feature-effects appearing consistently throughout the product variants.

Overall, there were a number of lessons learned from conducting this analysis on a real-world industrial product line:

- Considering additional sources of variability information improved the result of the analysis. The original prototypes for the feature-effect analysis on the PS-EC product line did not consider all the variability sources listed in Section 3.1. The variability conditions in interface descriptions and the list of legacy variability variables were only added later during the development process. The inclusion of the variability conditions from the interface descriptions improved the accuracy of the feature-effects; generally, the more sources where a variability variable may have an effect on the product line are considered, the more accurate the computed feature-effects become. The inclusion of the legacy variability, which effectively replaces some variables in the variability conditions with constants, helped to reduce the size of the resulting feature-effect formulas. This way, the formulas were more focused on relevant variability.
- The analysis in this chapter was carried out on a diverse set of products. This led to the results showing rather inconsistent classifications for the feature-effects of the variability variables. Analysis of groups of more similar product variants may lead to more consistent, and thus more usable result.
- Both, identifying dependent and independent/top-level variables can guide the creation of variability model constraints. Top-level variables, i.e. variables that always have an effect on the product, represent the highest level of the dependency hierarchy. The feature-effects of dependent variables can serve as a basis for the creation of variability model constraints, or they can be used to verify existing constraints. Variability model constraints have the potential to save effort and increase the quality of the configuration creation process. Additionally, they could also be used to guide the definition of new test cases by ensuring that configurations for the test-cases result in actually different products.
- The flexibility of KernelHaven allowed the analysis pipeline to be adapted to the PS-EC use-case. A preparation plug-in converted all integer-based variability to propositional logic, which can be handled by the data models of KernelHaven. New extractor plug-ins allowed the adaptation to the specifics how the PS-EC product line implements variability, including custom extractors for proprietary input file formats. The analysis components were extended to adapt the feature-effect approach to the specifics of the PS-EC product line; for example, the aggregation of artificial Boolean variables that were introduced as replacements for the integer-based variability variables was added to the feature-effect computation.

## 4 Extraction of Source Code Combined with Variability

This chapter presents the srcML-Extractor that was developed as a code model extractor for KernelHaven in the context of this thesis. It combines the traditional abstract syntax tree (AST) representation of source code with variability information. This enables analyses to augment well-established analysis techniques for traditional code with variability information. Additionally, the interaction of constructs in the source code with variability mechanisms can be analyzed. The use-case that initiated the development of this extractor is a framework for implementing variability-aware software product line metrics. Such metrics quantify aspects of the code, the variability, and the combination of both. This use-case and its motivation are introduced in Section 4.1. Section 4.2 introduces the srcML tool, that is used for the basic parsing of source code. The output of the srcML parsing requires further transformations to fulfill the requirements of the metric framework. This transformation process performed by the srcML-Extractor is explained in Section 4.3. Finally, Section 4.4 discusses alternative approaches that enable the analysis of source code combined with variability information.

### 4.1 Use-Case

A recent systematic literature review on variability-aware software product line metrics found shortcomings in the reviewed field [EYS18]. In contrast to traditional software metrics, variability-aware metrics consider the variability that is present in software product lines. The literature review found a number of metrics that are designed for software product lines. However, it also found the following weaknesses in the field of variability-aware metrics:

- There is a lack of metrics that combine information from the variability model and the source code. Most metrics are defined only for one or the other aspect. However, the variability in software product lines spans the whole project. Metrics that perform a more encompassing analysis that includes all facets of variability are likely beneficial.
- There is a lack of proper evaluations for variability-aware metrics. The literature review found that only about 36 % of the metrics had any evaluation at all. Additionally, some of the evaluations lacked in quality. Evaluations of metrics are important, as they explore how the application of the metrics can be beneficial in practice.
- There seems to be limited knowledge about existing variability-aware metrics in the community. The literature review found multiple instances where conceptually equal metrics were developed independently by different groups. This is disadvantageous, as it may lead to confusion about the multiply defined metrics. Additionally, the effort that went into the re-creation of existing metrics could be spared. The literature

review already remedies this to some extent, as it provides an encompassing overview of the existing variability-aware metrics.

- Established traditional (i.e. non-variability-aware) metrics were often not considered in the creation of new metrics. Concepts that are well-known in traditional metrics were not considered for the variability-aware metrics. The authors of the literature review argue that adapting traditional metrics to the variability information leads to a more comprehensive set of metrics.

MetricHaven, a framework for implementing variability-aware metrics, was conceived to address some of the findings of the systematic literature review [EKS19]. MetricHaven is implemented as a plug-in for the KernelHaven analysis framework [EK19]. It follows the basic architecture of KernelHaven, where specialized extractors create common data models that are passed to the metrics (cf. Section 2.4). The metrics are implemented on the generic interface of the common data models, thus they are independent from implementation details of specific product lines.

MetricHaven enables metrics to combine the variability information from different sources. This comes from the basic architecture of KernelHaven, where the three sub-pipelines extract different kinds of variability information. All these variability sources are available for the metric implementations. This enables the creation of metrics that combine information from the variability model and the source code, which was an issue detected by the systematic literature review. Additionally, MetricHaven allows the combination of metrics. For instance, variability model metrics can be used as weights in the calculation of variability-aware source code metrics. Such combinations of metrics lead to new mixed metrics, which combine information from the variability model and the source code.

Metrics that are implemented in the MetricHaven framework can easily be executed on different product lines. The KernelHaven framework supplies various utility functions and extractors that are required to run metrics on different software product lines. New metrics can be implemented with relatively little effort, as they do not have to handle common functionality that is already implemented in KernelHaven and MetricHaven. This gives researchers the tools to evaluate existing and new metrics. The literature review found that such evaluations are missing in many cases.

MetricHaven implements many of the metrics found in the systematic literature review. This serves as documentation of the metrics and as a basis for researches and practitioners to gather experience with them. This addresses the weakness found in the literature review, that there seems to be limited knowledge about existing metrics. MetricHaven, as an open-source project that implements most of the known variability-aware metrics, has the potential to increase the awareness about existing metrics.

Traditional metrics are also supported by MetricHaven. Established metrics on source code that do not consider variability may still be interesting in combination with the variability-aware metrics. Additionally, this addresses the finding of the literature review that new variability-aware metrics often have a weak connection to traditional ones. MetricHaven supports both kinds of metrics, thus it can help the creation of new variability-aware metrics based on traditional ones.

The Linux Kernel [Lin19] served as the use-case for the development of MetricHaven. This large software product line is available as open-source and is commonly used as an evaluation use-case in software product line research. The properties of the the KernelHaven architecture make MetricHaven agnostic to specific product lines, even though it

was developed for this specific use-case. The Linux Kernel has three kinds of variability that reflect the basic structure of the three sub-pipelines in KernelHaven:

**Kconfig** The variability model of the Linux Kernel is implemented in the Kconfig language. It defines the variability variables with a name, type, and description. Additionally, constraints between the variables define when they can be de-/selected. This variability model is extracted with the KconfigReaderExtractor in KernelHaven [KE18b].

**Kbuild** The Kbuild build system controls the compilation process of the Linux Kernel. It is based on Makefiles and decides, based on the configuration, which files are included in the build process. The variability information of this conditional inclusion of source files is extracted with the KbuildMinerExtractor in KernelHaven [KE18a].

**C-preprocessor blocks** In the C-source code files of the Linux Kernel, conditional compilation directives of the C-preprocessor are used. Variability variables are prefixed with `CONFIG_` to differentiate them from other preprocessor symbols. The srcML-extractor used in this sub-pipeline is developed in this chapter.

The extractors for the Kconfig and Kbuild systems already existed as plug-ins for KernelHaven. They comprehensively parse and represent the variability information from their respective assets. However, there was no adequate plug-in available for the source code extraction. A code extractor for this use-case is developed in the context of this thesis and presented in this chapter.

MetricHaven requires a combined view of the traditional abstract syntax tree (AST) and the variability information of the source code. For traditional metrics, i.e. metrics that do not consider variability, the traditional AST alone would suffice. For example, the Cyclomatic Complexity metric from McCabe requires the control flow graph which can be read from an AST [McC76]. However, variability-aware metrics additionally require the variability information (e.g. the hierarchy of conditional compilation blocks). While some of the variability-aware metrics require only the variability information, others combine the information from the traditional AST with the variability information. For instance, the Lines of Feature Code (LoF) metric counts how many source code lines are nested inside conditional compilation blocks [EYS18]. This can be implemented without the traditional AST. However, the Degree Centrality (DC) metric combines the nesting of conditional compilation blocks with the number of incoming and outgoing edges in the call graph [EYS18]. The former aspect of the DC metric requires the variability information and the latter aspect requires the traditional AST structure.

The main challenge of a code model extractor for MetricHaven is to parse the traditional AST structure and the variability information together. This also requires that a data model is developed that can represent both aspects at the same time. Traditional metric tools only extract the AST and cannot handle variability. In the context of software product lines, they can only be used to analyze a single product (cf. product-specific analysis in Section 2.3). On the other hand, the CodeBlockExtractor presented in Section 3.3 can only extract variability information. It does not parse the C-code and thus lacks important information for the metrics.

The model created by the extractor should also represent the developer perspective on the source code file. This means, that the elements in the model should closely reflect the structure that the developer sees when looking at the source code file. For instance, variability should not be resolved by enumerating the possible variations of the AST. Instead,

the C-preprocessor blocks should appear in the model as surrounding elements, as they do in the source code itself. This requirement stems from the use-case for MetricHaven, that investigates the influence of variability on the developers. Here, it is required to quantify the variability that the developer has to work with in the source code. A model that does not reflect the structure of the variability in the source code would hinder this goal.

## 4.2 srcML Parsing Tool

The srcML-extractor presented in this chapter uses the srcML tool [CDM11; CMD+17] for the basic parsing of C-source code. The term srcML refers to both, the tool that parses source code and the resulting markup structure (“source markup language”). The srcML tool parses the C-code and adds the result as XML markup on-top of the source code. The term srcML-extractor refers to the KernelHaven plug-in that internally uses the srcML tool for the creation of the code model.

```
1
2 int main() {
3     int a = 1;
4     if (a >= 1) {
5         return 1;
6     }
7     return a;
8 }
```

(a) C-source code

```
1 <unit xmlns="http://www.srcML.org/srcML/src"
  ↪ xmlns:cpp="http://www.srcML.org/srcML/cpp" revision="0.9.5"
  ↪ language="C" filename="file.c">
2 <function><type><name>int</name></type>
  ↪ <name>main</name><parameter_list>()</parameter_list> <block>{
3   <decl_stmt><decl><type><name>int</name></type>
  ↪ <name>a</name> <init>=<expr><literal
  ↪ type="number">1</literal></expr></init></decl>;</decl_stmt>
4   <if>if <condition>(<expr><name>a</name>
  ↪ <operator>&gt;=</operator> <literal
  ↪ type="number">1</literal></expr></condition><then> <block>{
5     <return>return <expr><literal
  ↪ type="number">1</literal></expr>;</return>
6   }</block></then></if>
7   <return>return <expr><name>a</name></expr>;</return>
8 }</block></function>
9 </unit>
```

(b) Parsed srcML

Figure 4.1: Example of C-source code parsed by srcML

Figure 4.1 shows an example of how srcML represents its parsing result. Figure 4.1a shows the original C-source code that is parsed. Figure 4.1b shows the parsing result created by the srcML tool. All tokens of the original code are still present in the XML structure. The XML tags added around them represent the syntactic structure of the code. For instance,

the tokens in line 7 are wrapped in a `<return>` tag, which indicates that this construct is a return statement. Inside of this construct, the token `a` is marked-up to be an identifier (`<name>`) inside the return expression (`<expr>`).

```

1 <unit xmlns="http://www.srcML.org/srcML/src"
  ↪ xmlns:cpp="http://www.srcML.org/srcML/cpp" revision="0.9.5"
  ↪ language="C" filename="file.c">
2   <function>
3     <type><name>int</name></type>
4     <name>main</name>
5     <parameter_list>()</parameter_list>
6     <block>{
7       <decl_stmt>
8         <decl>
9           <type><name>int</name></type>
10          <name>a</name>
11          <init>
12            =
13            <expr>
14              <literal type="number">1</literal>
15            </expr>
16          </init>
17        </decl>
18      <decl_stmt>
19      <if>
20        if
21        <condition>(
22          <expr>
23            <name>a</name>
24            <operator>&gt;=</operator>
25            <literal type="number">1</literal>
26          </expr>
27        )</condition>
28        <then>
29          <block>{
30            <return>
31              return
32              <expr>
33                <literal type="number">1</literal>
34              </expr>
35            <return>
36          }</block>
37        </then>
38      </if>
39      <return>
40        return
41        <expr><name>a</name></expr>
42      <return>
43    }</block>
44  </function>
45 </unit>

```

Figure 4.2: Formatted parsing result of srcML

The srcML markup retains the complete text content of the original code [CDM11]. If the XML tags were removed, the original code is completely re-created. This also includes newlines and whitespace characters; all tokens inside the XML have the same line number as in the original source code. This makes the resulting XML hard to read. Figure 4.2 shows the same parsing result as Figure 4.1b, formatted for readability. This shows that the srcML markup is an abstract syntax tree (AST) of the code. For instance, the function `main` (starting in line 2) has a `<type>`, `<name>`, and `<parameter_list>` followed by a function body (`<block>`) with nested statements. This equals the data structure used to represent a function in an AST. srcML arranges its AST in a way that allows it to keep the original tokens in their original order and location.

In contrast to a compiler, srcML can also parse and add markup for incomplete or syntactically invalid programs. srcML aims at a low-level representation of the AST [Col05]. Higher-level information, such as types and call graphs, are not included. srcML only adds low-level syntactical markup for the tokens in the source code. For instance, any statement starting with the keyword `return` up to the semicolon that ends the statement is marked-up with `<return>`. srcML does not add information which function this return statement belongs to (see for example the return statements in Figure 4.1). This approach of marking-up the tokens with low-level syntactical information makes the srcML tool more accepting than a compiler.

srcML has a developer-centric view on the source code [CDM11], that is, srcML represents the code in a way that is very similar to its structure in the source code file. This is a result of the low-level markup approach of srcML: the basic structure of the code is retained and is only enriched by additional syntactical markup. In addition to this, srcML also does not evaluate C-preprocessor directives before parsing the code. Instead, the C-preprocessor directives are included in the XML markup of the file. This also supports the developer-centric view on the source.

```

1  a =
2      #ifdef A
3          1
4      #else
5          2
6      #endif
7  ;
    
```

(a) C-source code with C-preprocessor directives

```

1  <expr_stmt><expr><name>a</name> <operator>=</operator>
2      <cpp:ifdef>#<cpp:directive>ifdef</cpp:directive>
   ↪ <name>A</name></cpp:ifdef>
3          <literal type="number">1</literal>
4      <cpp:else>#<cpp:directive>else</cpp:directive></cpp:else>
5          <literal type="number">2</literal></expr>
6      <cpp:endif>#<cpp:directive>endif</cpp:directive>
   ↪ </cpp:endif>
7  ;</expr_stmt>
    
```

(b) Parsed srcML with C-preprocessor directives

Figure 4.3: Example of C-source code with C-preprocessor directives parsed by srcML

Figure 4.3 shows an example how C-source code with C-preprocessor directives is parsed. The `#ifdef` in line 2 is marked up as an `<cpp:ifdef>` and inserted in the XML in the location that corresponds to its location in the source code. There is no higher-level information that the `#ifdef`, `#else`, and `#endif` belong to each other. The three C-preprocessor elements are simply added into the `<expr>` and `<expr_stmt>` elements, which corresponds to their placement in the source code.

The developer-centric view of the srcML markup makes it ideal for the use-case of MetricHaven. As explained in Section 4.1, the code model of MetricHaven should represent the developer perspective on the source code. srcML fulfills this requirement by keeping the complete structure of the original source code file and only adding syntactical markup that preserves this structure. The syntactical information forms an AST, which can be used for the metrics that require the traditional AST structure. The variability information is retained, as the C-preprocessor directives are not resolved, which can be used for the metrics that require the variability information.

While the basic structure of the srcML markup fits the use-case of MetricHaven, there are also some adaptations required. First, the srcML markup is created as an XML structure. This needs to be converted into a Java object structure that is compatible with the code model interfaces of KernelHaven. This is required as all extractors in KernelHaven need to conform to the common data model interfaces (cf. Section 2.4). Additionally, the integration of the conditional compilation blocks of the C-preprocessor needs to be transformed. As shown above in Figure 4.3, the `#ifdef` directives are simply marked up to be `#ifdef` directives; there is no information that the elements between the `#ifdef` and the `#else` are actually nested inside the `#ifdef`. However, the resulting code model should reflect that the elements after the `#ifdef` are nested inside a conditional compilation block. This requires a transformation of the structure that srcML creates.

### 4.3 Transformation of srcML

This section describes the main implementation part of the srcML-extractor plug-in for KernelHaven [KE19b]. The extractor uses the srcML tool for the basic parsing of source code files. After that, a transformation process is required that adapts the parsing results of srcML to adhere to the code model interface required by KernelHaven. This section describes this transformation process.

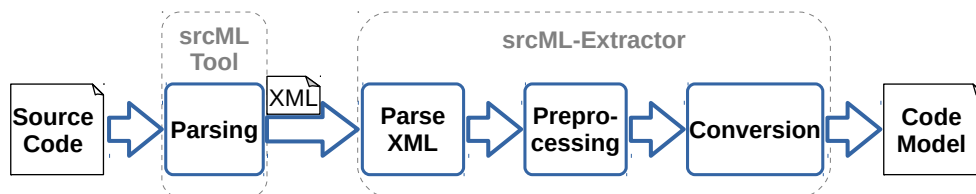


Figure 4.4: srcML-Extractor transformation process

Figure 4.4 shows an overview of the transformation process. The original source code file on the left is first parsed by the srcML tool. This results in an XML markup as shown in Section 4.2. This XML markup is the input for the srcML-extractor. After parsing the XML using the standard Java API, the XML structure is first passed through a preprocessing step. This step performs modifications on the XML nesting structure that are required before further conversion. After this preprocessing step, the XML structure is converted to

an AST that implements the KernelHaven code model interface. This results in the code model that is then passed to the analysis (i.e. metrics in the case of MetricHaven).

The following subsections will explain the preprocessing and conversion steps in detail. Subsection 4.3.1 describes the preprocessing step that transforms the nesting structure of `#ifdef` directives. Subsection 4.3.2 describes the AST that implements the required KernelHaven interface and the conversion of the XML structure to this AST structure.

### 4.3.1 Transformation of Variability Nesting Structure

The srcML tool only marks-up C-preprocessor conditional compilation directives (`#ifdef` etc.) and does not create a nesting structure for them. This can be seen in Figure 4.5, which shows a simplified version of the srcML markup from Figure 4.3. The `<literal>` in line 4 is not nested inside the `#ifdef` in line 3, although semantically it should be. The C-preprocessor directives are only added as a flat structure into the XML. Additionally, there is also no information that the `#ifdef`, `#else`, and `#endif` belong to the same construct. This is because srcML does not parse such higher-level information; it only does low-level syntactical markup (cf. Section 4.2).

```
1 <expr>
2   <!-- ... -->
3   <cpp:ifdef>#ifdef A</cpp:ifdef>
4   <literal>1</literal>
5   <cpp:else>#else</cpp:else>
6   <literal>2</literal>
7   <cpp:endif>#endif</cpp:endif>
8   <!-- ... -->
9 </expr>
```

Figure 4.5: C-preprocessor markup created by srcML (simplified)

The code model created by the srcML-extractor should reflect that elements of the AST are nested inside conditional compilation blocks. MetricHaven requires a combined view of the variability and the traditional AST structure. The srcML approach that inserts C-preprocessor directives individually without considering nesting is not enough. The variability information has to be *combined* with the AST structure. Figure 4.6 shows the (simplified) nesting structure that combines the variability information with the AST. Here, the `<literal>` elements are nested inside the C-preprocessor blocks.

```
1 <expr>
2   <!-- ... -->
3   <cpp:ifdef condition="A">
4     <literal>1</literal>
5   </cpp:ifdef>
6   <cpp:else condition="!A">
7     <literal>2</literal>
8   </cpp:else>
9   <!-- ... -->
10 </expr>
```

Figure 4.6: Required C-preprocessor nesting (simplified)

The preprocessing step in the srcML-extractor transformation process (cf. Figure 4.4) transforms the flat C-preprocessor markup created by srcML to the required nesting structure. This is performed on the XML before it is converted to the AST objects in the next step. First, the preprocessing matches which conditional compilation statements belong together. For instance, in Figure 4.5 this would identify that the `#ifdef` in line 3 is closed by the `#else` in line 5; the `#else` is closed by the `#endif` in line 7. This matching also considers that conditional compilation blocks can be nested inside each other, by using a stack to keep track of the nesting level. If conditional compilation blocks are missing siblings (e.g. an `#endif` without an `#ifdef`) then the C-preprocessor information is malformed and cannot be parsed by the srcML-extractor. In this case, parsing of the whole file is aborted.

After all matching conditional compilation statements are found, the nesting structure is transformed. The basic approach is to move all XML nodes that appear between the opening and closing conditional compilation statements into the opening XML node. For instance, the `<literal>` in line 4 of Figure 4.5 appears between the opening `#ifdef` in line 3 and its closing `#else` in line 5. Thus, it is moved into the `<cpp:ifdef>` node as seen in Figure 4.6. The `#else` itself opens another block where the `<literal>` from line 6 is moved into. The `#endif` in line 7 only ends the previous `#else` block. Since the nesting structure now fully reflects which elements are nested inside the conditional compilation blocks, the `<cpp:endif>` node is removed.

Note that in the simplified figures used to illustrate this example the locations of the XML nodes change. For example, the `<literal>` that is originally in line 6 of Figure 4.5 is moved to line 7 of Figure 4.6. This is only because the figures are simplified and formatted for readability; the actual implementation takes care that location information is fully preserved. This allows the code model that is eventually created in the srcML-extractor to have correct references to the original locations in the source file. Also, the C-preprocessor tokens (e.g. the `#ifdef A`) in Figure 4.5 have been transformed to `condition` attributes in Figure 4.6. This is only done for illustration purposes and does not reflect the implementation of the srcML-extractor.

Figure 4.5 does not exactly reflect the nesting structure of Figure 4.3. The actual nesting structure is shown in Figure 4.7. The `<cpp:endif>` is not nested inside the `<expr>` that the other C-preprocessor directives are nested in, but as a sibling of the `<expr>` statement. This is a commonly occurring scenario in the output of the srcML tool. srcML detects that the expression is finished after the last expression element (the second `<literal>`) and closes the `<expr>` tag. The `<cpp:endif>` is inserted after this, even though all its corresponding siblings are inside the `<expr>`. A similar scenario can happen with an opening `#ifdef` at the start of a statement or expression: srcML first inserts the `<cpp:ifdef>` node, before the statement or expression node is opened. These scenarios occur because srcML only does low-level markup and does not consider where the siblings of a C-preprocessor directive are located.

The scenarios, where the opening or closing conditional compilation directive are on the wrong nesting level, cause problems for the transformation process. For instance, in Figure 4.7 there is now a closing `</expr>` node in line 8 between the `<cpp:else>` and the `<cpp:endif>`. Moving all nodes that are enclosed by the `<cpp:else>` and `<cpp:endif>` is not possible anymore, as the closing `</expr>` node cannot be moved without its corresponding opening node in line 2. For this kind of scenario, the srcML-extractor has a special correction mechanism: if such a scenario is detected, the nesting structure is adapted so that the normal process can continue. In the case of Figure 4.7, the `<cpp:endif>` in line 9

```

1  <expr_smt>
2      <expr>
3          <!-- ... -->
4          <cpp:ifdef>#ifdef A</cpp:ifdef>
5              <literal>1</literal>
6          <cpp:else>#else</cpp:else>
7              <literal>2</literal>
8      </expr>
9      <cpp:endif>#endif</cpp:endif>
10 ;</expr_smt>
    
```

Figure 4.7: Problematic C-preprocessor markup created by srcML (simplified)

is moved into the `<expr>`, as for example shown in Figure 4.5. The normal transformation of the nesting structure of the C-preprocessor statements can then proceed.

Another problematic scenario occurs when the conditional compilation structure causes the AST structure to change depending on the variability. This can happen, for instance, if the header of an if-statement is inside an `#ifdef`, but the body of that if-statement is not. Figure 4.8 illustrates this problem. If the `#ifdef` is selected, the block with the statement has to be nested inside the if-statement. However, if the `#ifdef` is deselected, the block with the statement has to be nested directly under the parent root node. This creates a problem for the code model that the srcML-extractor creates: both possible variations have to be represented in the AST, but the block with the statement can only have one parent in the AST.

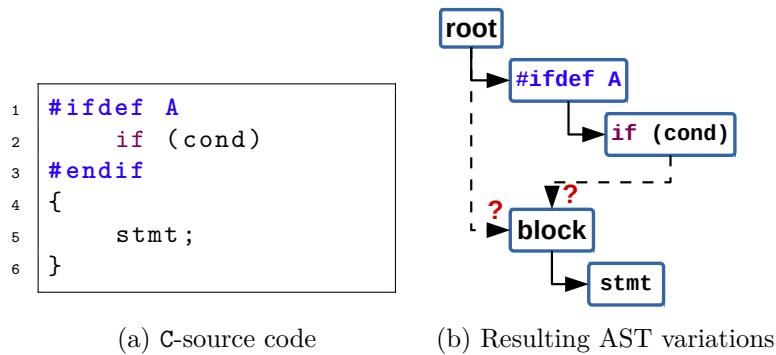


Figure 4.8: Code with changing structure depending on conditional compilation

Figure 4.9 shows the (simplified) markup that the srcML tool generates for the code in Figure 4.8a. The srcML tool parses the code as if there were no C-preprocessor directives. Thus, the block with the statement is marked-up to belong to the if-statement. However, the srcML-extractor cannot simply keep this structure as this does not adequately reflect the case that the block with the statement can also occur outside of the `#ifdef`. For this kind of scenario, the srcML-extractor has a special mechanism that is applied when three conditions are met:

- (a) the opening conditional compilation directive (the `<cpp:ifdef>` in this case) is in front of a structure (the `<if>` in this case), and
- (b) the closing conditional compilation directive is somewhere nested inside that structure (the `<cpp:endif>` is nested inside the `<if>`), and

```

1 <cpp:ifdef>#ifdef A</cpp:ifdef>
2 <if>if
3     <condition>(cond)</condition>
4     <then>
5         <cpp:endif>#endif</cpp:endif>
6         <block>{
7             <statement>
8                 stmt;
9             </statement>
10        }</block>
11    </then>
12 </if>

```

Figure 4.9: srcML markup of Figure 4.8a (simplified)

- (c) there are further elements after the closing conditional compilation directive nested inside the structure (in this case, the `<block>` follows the `<cpp:endif>`).

When all three conditions are met, the srcML-extractor preprocessing transforms the XML structure so that both possible variations of the AST are reflected. The closing conditional compilation block and the following nodes (i.e. the `<cpp:endif>` and `<block>` nodes in Figure 4.9) are moved behind the structure that they were previously nested in (the `<if>` in this case). In the place of the nodes following the conditional compilation block (the `<block>` node), special reference nodes are created that indicate that the nodes may occur in either place of the AST.

```

1 <cpp:ifdef>#ifdef A</cpp:ifdef>
2 <if>if
3     <condition>(cond)</condition>
4     <then>
5         <reference refid="1" />
6     </then>
7 </if>
8 <cpp:endif>#endif</cpp:endif>
9 <block id="1">{
10     <statement>
11         stmt;
12     </statement>
13 }</block>

```

Figure 4.10: Transformation result with reference node (simplified)

Figure 4.10 shows the result of such a transformation for the example in Figure 4.9. The `<cpp:endif>` and `<block>` are moved behind the closing `</if>` node. In the original place of the `<block>` node (line 5), a special `<reference>` node is created that refers to the `<block>` node.<sup>1</sup> This XML structure closely reflects the information that is present in the original source code (Figure 4.8a). The if-statement is enclosed by an `#ifdef`. The

<sup>1</sup>For this illustration, the `<block>` node got an `id` attribute and the `<reference>` node got a `refid` attribute with the same ID. However, in the actual implementation of the srcML-extractor, the reference is implemented in a different way (using the `UserData` API of the Java XML API). The different approach using IDs in this illustration was chosen for better readability in text-form.

following block with a statement is not inside that `#ifdef`. However, the if-statement has a reference to the block, as it represents the body of the if-statement if it is compiled-in.

While the standard process and the two specialized mechanisms for transforming the variability nesting structure can handle many common cases of variability, there are C-preprocessor constructs possible that cannot be handled by the srcML-extractor. This is because the C-preprocessor does arbitrary text-based modifications on the source code. This can lead to constructs where the conditional compilation blocks cannot be integrated with the AST structure of the source code. If this is the case, parsing of the whole file is aborted; the srcML-extractor is not able to create a model where the AST and the variability information are combined.

```
1  if (
2  #ifdef A
3      cond1)
4      stmt1;
5  #else
6      cond2)
7  #endif
8      stmt2;
```

Figure 4.11: Example C-preprocessor construct that cannot be transformed

Figure 4.11 shows a C-preprocessor construct that cannot be transformed by the srcML-extractor. The problem here is that the `#ifdef` and the `#else` blocks cover different elements of the AST. The `#ifdef` contains the condition of the if-statement (line 3) and the then-statement (line 4), while the `#else` block only contains the condition of the if-statement (line 6). If both blocks would only contain an if-condition (and the then-statement would not be nested inside a C-preprocessor block), then the conditional compilation blocks could be inserted into the AST. However, it is not possible with the AST structure of the srcML-extractor that the `#ifdef` block surrounds both, the condition and the then-statement of the if-statement. Technically, the transformation process of the C-preprocessor blocks fails because the opening `#ifdef` in line 2 starts nested inside the if-condition, while its closing `#else` appears on a higher level after the then-statement.

This problem of C-preprocessor directives that do not integrate into the AST structure is known as *undisciplined preprocessor annotations* [LKA11]. Liebig et al. define disciplined annotations as C-preprocessor directives that cover whole statements or functions. Consequently, undisciplined C-preprocessor directives are inside statements, or break up structures like if-statements and function definitions. The code in Figure 4.11 falls into this definition of undisciplined annotations. Liebig et al. found in an empirical analysis of 40 software projects, that while most C-preprocessor directives are disciplined, there are undisciplined occurrences in almost all projects. Medeiros et al. propose a catalog of refactorings to turn undisciplined C-preprocessor directives into disciplined ones [MRG+17]. They also found that developers are generally in favor of disciplined C-preprocessor directives, as opposed to undisciplined ones.

Another problem in the handling of C-preprocessor blocks is caused by the way the srcML tool parses code around C-preprocessor blocks. srcML parses the C-tokens as if there were no C-preprocessor directives. The problem that results from this is illustrated in Figure 4.12. On the left (Figure 4.12a), the source code shows two alternative then-statements (lines 4 and 8) for an if-statement (line 1). On the right (Figure 4.12b), the (simplified)

parsing result of the srcML tool is shown. Lines 5 and 6 show that the then-block and the if-statement are closed after the first possible then-statement in line 4. The srcML tool ignores the C-preprocessor directives and thus cannot parse line 8 as another possible then-statement; the second statement (line 8) is not included in the if-statement.

<pre> 1  if (cond) 2 3  #ifdef A 4      stmt1; 5 6 7  #else 8      stmt2; 9  #endif </pre>	<pre> 1  &lt;if&gt;if &lt;condition&gt;(cond)&lt;/condition&gt; 2      &lt;then&gt; 3          &lt;cpp:ifdef&gt;#ifdef A&lt;/cpp:ifdef&gt; 4              &lt;expr_stmt&gt;stmt1;&lt;/expr_stmt&gt; 5          &lt;/then&gt; 6  &lt;/if&gt; 7  &lt;cpp:else&gt;#else&lt;/cpp:else&gt; 8  &lt;expr_stmt&gt;stmt2;&lt;/expr_stmt&gt; 9  &lt;cpp:endif&gt;#endif&lt;/cpp:endif&gt; </pre>
(a) C-source code	(b) Markup by srcML (simplified)

Figure 4.12: Example C-preprocessor construct that srcML marks-up incorrectly

In cases like this, where the srcML tool does not detect alternatives and instead only considers the first element, the resulting AST structure does not correctly represent the source code content. Only the first element is marked-up with the correct semantic information, while the second element is considered to occur after the construct in question. In Figure 4.12 this means that only the first statement is marked-up to be a then-statement for the if-statement; the second statement appears after the if-statement. Since this error in the markup is created by the srcML tool, it is carried through the srcML-extractor transformation process. Such errors will still appear in the final code model that the srcML-extractor creates.

In summary, the srcML-extractor can correctly transform the nesting structure for all *disciplined* C-preprocessor directives as defined by Liebig et al. This makes up most of the variability in real-world software projects. *Undisciplined* C-preprocessor directives can often be transformed via two specialized mechanisms: fixing of the nesting location for opening and closing directives, and introducing reference nodes. However, there are also *undisciplined* C-preprocessor directives that cannot be handled as they go against the AST structure. Additionally, some *undisciplined* C-preprocessor directives that are transformed by the srcML-extractor do not reflect the original source code as the srcML tool creates incorrect mark-up.

### 4.3.2 XML to AST Conversion

The final step in the transformation process of the srcML-extractor is the conversion of the XML to an AST structure that implements the KernelHaven code model interface (cf. Figure 4.4). The basic concept in this step is to stay close to the structure that is created by srcML. In many cases, nodes in the XML structure are converted directly to Java objects. For this, a class hierarchy that can represent the AST structure and implements the KernelHaven code model interface is implemented.

Figure 4.13 shows an excerpt of the class diagram for the AST structure of the srcML-extractor. The **SourceFile** class and the **CodeElement** interface (marked orange) are the required structure of the code model for KernelHaven. All code model extractors have to create a data structure that implements this interface. A **SourceFile** is an aggregation

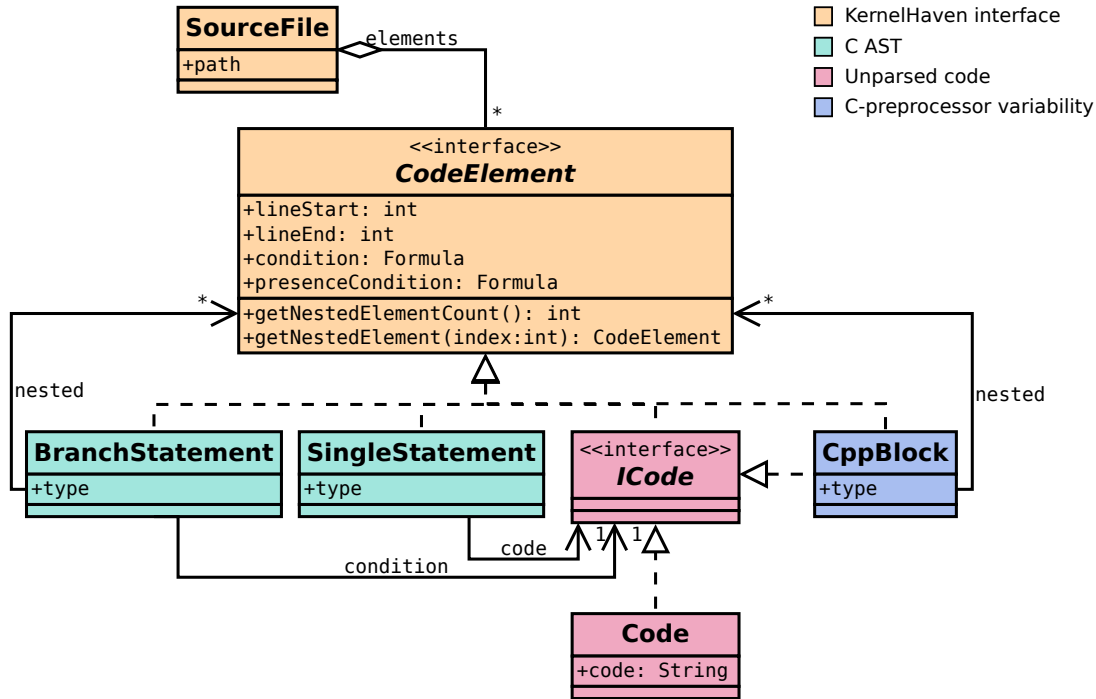


Figure 4.13: AST class diagram of srcML-extractor (simplified excerpt)

of `CodeElements`. A `CodeElement` represents an element in the source code file. It has a location (start and end line number), an immediate condition (the condition of the closest surrounding C-preprocessor block), and a presence condition (cf. Section 3.3). Further `CodeElements` can be nested inside a `CodeElement`. In the AST structure of the srcML-extractor, all elements in the AST implement this interface.

The `BranchStatement` and `SingleStatement` in Figure 4.13 represent an excerpt of the classes that are used to build the C-AST (marked turquoise). The `BranchStatement` is used to represent if-, else-if-, and else-statements. The `SingleStatement` class is used to represent all kinds of statements that stand on their own: among others, the `<expr_stmt>`, `<continue>`, `<break>`, and `<return>` XML nodes created by srcML are converted to `SingleStatements`. As different XML nodes in the srcML markup are converted to the same class, the classes have a `type` attribute that allows to differentiate between the different semantic meanings. The `BranchStatement` has a list of nested elements that represent the body of the branch, e.g. the then-statements of an if-statement. The class structure allows arbitrary `CodeElements` as nested elements, although semantically only statements are allowed here. This follows the lenient approach of the srcML tool, which allows constructs in the AST which represent invalid programs.

The `CppBlock` class in Figure 4.13 represents conditional compilation blocks of the C-preprocessor (marked blue). Its `type` attribute represents the type of directive for the block (e.g. `#ifdef` or `#else`). It implements the `CodeElement` interface, so it can be inserted in arbitrary positions in the nesting structure of the AST. This is an advantage of the lenient approach, that allows arbitrary `CodeElements` to be nested in parent elements (as seen above with the `BranchStatement`). The conditions and presence conditions for C-preprocessor blocks are calculated following the same rules as defined in Section 3.3. The srcML-extractor re-uses the expression parsing component of the `CodeBlockExtractor`.

The AST structure created by the srcML-extractor only represents the srcML markup up to a statement-level of granularity. The more fine-grained level of expressions is not represented as an AST, but rather kept as an unparsed code string. For instance, the condition of a **BranchStatement** and the content of a **SingleStatement** are represented as such unparsed text. In the class structure in Figure 4.13, this is implemented via the **ICode** interface and the implementing class **Code** (marked red). The **Code** class has a string attribute that contains unparsed code. Unparsed code may contain C-preprocessor blocks, which need to be represented in the AST. This is realized by the **CppBlock** class implementing the **ICode** interface. This way, conditional compilation blocks in expressions can be accurately represented by **CppBlocks**, while the content of the expression remains as unparsed code. If a **CppBlock** appears in an expression, the text content can be split into multiple **Code** elements to accurately represent which parts of the condition are nested in the **CppBlock**.

The main driver for the decision to limit the granularity of the srcML-extractor AST was the reduction of implementation effort. A large portion of an AST representation for the C-programming language is dedicated to representing the expressions. This includes operators, literals, function calls, etc. The effort to implement the required AST structure and the conversion routines for the corresponding srcML nodes was spared in the implementation of the srcML-extractor. Additionally, this approach makes the srcML-extractor more accepting, as there can be no conversion errors for the code that is only kept as text. This was possible as almost no metrics in MetricHaven, which is the use-case for the srcML-extractor, require information from the AST beyond the statement-level. The metrics that do require information from inside expressions can be implemented heuristically on the unparsed code string.

The preprocessing step that converts the variability nesting may insert **<reference>** nodes to transform some C-preprocessor constructs (cf. Section 4.3.1). These **<reference>** XML nodes are converted to the **ReferenceElement** class in the srcML-extractor AST. This class has an attribute **referenced** that points to the referenced element in the AST. However, in the primary nesting structure of the AST (i.e. the **getNestedElement\*()** methods), the referenced element is not a child of the **ReferenceElement**. If this was the case, the referenced element would appear twice in the AST hierarchy. Instead, the **referenced** attribute constitutes a non-primary relation across the AST.

In some cases, C-preprocessor blocks can lead to constructs in the srcML structure that cannot be converted to the srcML-extractor AST. This is because the attributes of some AST elements require special children in the XML structure. For instance, a **<function>** node in the srcML markup has to have a **<name>** child, which is stored in the **name** attribute in the srcML-extractor AST. However, due to C-preprocessor blocks, the **<name>** child may not appear directly inside the **<function>** node. Also, multiple **<name>** nodes may appear for a single function. In these cases, the **name** attribute for the function in the srcML-extractor AST cannot be determined; the srcML-extractor cannot convert this part of the AST.

For cases where the srcML-extractor cannot convert the srcML markup, it implements a graceful back-off strategy. As opposed to errors in the preprocessing step that converts the variability nesting (cf. Section 4.3.1), the parsing of the whole file is not aborted. Instead, a special **ErrorElement** is inserted in the AST to represent the XML node that cannot be converted. After this, normal conversion for its children is resumed. This way, the rest of the file, including the children of the problematic XML node, can still be converted. When an **ErrorElement** is inserted, a flag is set for all parent elements (recursively) that

indicates that they contain an **ErrorElement**. If analyses require assurance that the code is parsed correctly, they can check this flag which indicates whether any nested element was not parsed correctly. This makes the srcML-extractor more accepting, and leaves the decision on the required accuracy for the analysis.

## 4.4 Alternative Approaches

This section introduces three alternative approaches that build models of source code containing variability. All of the approaches have in common that they enable variability-aware analysis of the source code. The use-case of this chapter, the MetricHaven framework for implementing variability-aware software product line metrics, is such a variability-aware analysis. It requires a code model that includes traditional syntactical information about the code and information about the variability. The three presented alternative approaches will be compared to the srcML-extractor, with respect to the application for the MetricHaven use-case.

As part of the **TypeChef** project, a variability-aware parsing approach for programming languages, including C-code with C-preprocessor variability, was developed [KGR+11]. The basic concept of this approach is to create an abstract syntax tree (AST) representation of the source code, which contains choice nodes that represent the variability. This data structure allows different variability-aware analyses to be conducted, including type-checking.

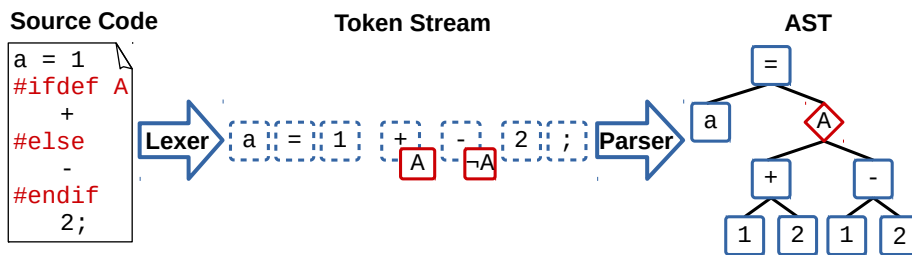


Figure 4.14: Parsing approach of TypeChef

The parsing of TypeChef is split into two components, a lexer and a parser. Figure 4.14 illustrates this: the lexer creates a stream of tokens that is passed to the parser. The lexer is variability-aware, that means it considers C-preprocessor directives while reading the C-source code file. It evaluates the C-preprocessor directives; for instance, file inclusions and macro expansions are performed before the token stream is created. In the case of conditional compilation blocks, the lexer annotates the tokens inside the block with a presence condition. The tokens of all alternatives are included in the stream. Tokens that are not nested in conditional compilation blocks have the presence condition *true*. This can be seen in the token stream in Figure 4.14: the `+` token is inside an `#ifdef A` block, thus it has the presence condition *A*. Similarly, the `-` token has the presence condition  $\neg A$  as it is inside the `#else` block. All other tokens have the presence condition *true*, which is omitted in this illustration.

The parser converts the token stream into an AST representation. Variability is represented by choice nodes in the AST. In the AST at the right-hand side of Figure 4.14, the choice node is represented as a red diamond. If the variability variable *A* is *true*, the left sub-tree with the `+` operator is used; if *A* is *false*, the right sub-tree with the `-` operator is used. When the parser encounters tokens with a new presence condition, different parsers are

created for the possible variants. In the example of Figure 4.14, the parser encounters the `+` token with the presence condition *A*. Two parser are created, one for the variant where *A* is *true* and one for the variant where *A* is *false*. The first parser creates the sub-tree with the `+` operation, the second parser creates the sub-tree with the `-` operation. Both parsers use the literal tokens 1 and 2. At the `;` token, the two parsers are joined together as the two variants converge again. In the AST, the choice node represents the location where the two parsers where split, with both child sub-trees as the results of the individual parsers. When splitting parsers, the constraints of the variability model are used to exclude parsers for variants that are not allowed. This reduces the number of necessary splits.

The TypeChef parser creates a syntactically sound and complete AST for the source code. In contrast, the srcML-extractor and its underlying srcML tool have a more lenient approach, which even allows them to parse some syntactically incorrect programs. The AST created by the TypeChef parser allows for sophisticated analysis of the source code, like type-checking. However, this requires that the AST is built with relatively strict constraints. For instance, variability is not inserted where it occurs in the original source code file, but rather where the parser requires it. Because of this, the AST only *semantically* reflects the content of the original source code file. In the AST in Figure 4.14, the variation point for the `#ifdef` in the source code is in a different location than in the original source code. The literals 1 and 2 occur multiple times in the AST, while they each only occur once in the source code. In contrast, the srcML-extractor builds a model that faithfully reflects the locations and structures in the source code file, including variability.

The MetricHaven use-case requires a code model that closely reflects the contents of the source code file (cf. Section 4.1). The TypeChef parser does not fulfill this requirement. The location and structure of C-preprocessor blocks is lost during parsing and is only vaguely represented by choice nodes. For metrics that are defined on the conditional compilation block structure, the representation of TypeChef is not useful. Additionally, some tokens or even entire sub-trees are duplicated to build a syntactically sound AST. The srcML-extractor creates a code model that preserves all C-preprocessor directives and integrates them with the traditional AST structure in a way that closely reflects the original source code file. This is a better fit for the MetricHaven use-case.

The **configuration lifting** approach creates a *meta-program* that represents all variants of the original program containing variability [PS08]. The underlying idea is to integrate the variability information into source code by representing them with the standard programming language structures. For instance, conditional compilation directives of the C-preprocessor are converted into standard if-statements of C. The configuration lifting approach thus converts *compile-time* variability into *run-time* variability.

The meta-program created by the configuration lifting approach enables existing, non-variability-aware static analyses to be conducted. As all variability mechanisms have been converted to constructs in the programming language, the static analysis tools can parse and analyze the meta-program without modification. This way, the static analysis tool can analyze all possible variants at once. The results found by the static analysis in the meta-program can be transferred to the underlying source code containing variability.

Not all conditional compilation blocks of the C-preprocessor can be trivially converted to if-statements in C-code. For example, Figure 4.15a shows an excerpt of a C-program where the type of the variable `x` depends on the variability variable `CONFIG_A` (lines 1 through 6). Figure 4.15b shows the meta-program that is created by the configuration lifting approach. Both declaration variants are present in the meta-program as different variables (lines 2

<pre> 1  #ifdef CONFIG_A 2  int 3  #else 4  float 5  #endif 6      x = 3; 7 8  int y = x + 2; 9 10 11 12                 </pre>	<pre> 1 2  int x_CONFIG_A = 3; 3 4  float x_NOT_CONFIG_A = 3; 5 6 7 8  int y = 9      (CONFIG_A ? 10         x_CONFIG_A 11         : x_NOT_CONFIG_A) 12         + 2;                 </pre>
---	---

(a) C-source code

(b) Resulting meta-program

Figure 4.15: Non-trivial lifting of C-preprocessor variability

and 4). The postfixes indicate which configuration they correspond to. The variability variable `CONFIG_A` is available as a run-time variable in the meta-program. The reference to variable `x` in the statement in line 8 is transformed to a conditional reference: the ternary operator is used to select, based on `CONFIG_A`, which variant of `x` is referenced. In this meta-program, a compiler or type-checker would find an error: `x_NOT_CONFIG_A` is of type `float`, but the statement in line 8 tries to assign it to variable `y` of type `int`.<sup>2</sup>

The meta-program created by the configuration lifting approach does not fulfill the requirement of MetricHaven that the source file content should be accurately represented. The information what elements in the code constitute variability is lost, as all variability is converted to standard structures in the programming language. In the meta-program, conditional compilation directives cannot be distinguished from standard if-statements. For metrics that are defined on the hierarchy of conditional compilation blocks, this representation is not useful. Additionally, the creation of a syntactically sound meta-program requires modifications to the code that diverge from the content of the original source file. This can be seen in Figure 4.15: the `#ifdef` disappears and the declaration and assignment statement of `x` is duplicated. In contrast, the srcML-extractor creates a model that more accurately represents the content of the original source code file.

A **variability-aware code property graph** builds a model of the source code by extending code property graphs with variability information [GS19]. Code property graphs combine three graph representations of source code: AST, control flow graph, and program dependence graph [YGA+14]. Nodes in a code property graph represent elements of the source code. Edges represent relations between the source code elements. There are edges that represent the AST structure (e.g. is-parent-of relations), control flow edges, and program dependence edges. Such a graph representation contains more information than the relatively simple low-level syntactical markup created by the srcML tool.

Gerling et al. extend the traditional code property graphs with variability information [GS19]. Additional nodes are introduced that represent the C-preprocessor directives used to implement variability. A new kind of edge connects the variability information with the

<sup>2</sup>Strictly speaking, this is not an error as C allows implicit conversions from `float` to `int`. gcc only prints a warning when the flag `-Wconversion` is set. For this illustration this distinction is not relevant. The point is that existing, non-variability-aware static analyses can find (potential) problems in the meta-program.

code elements. For instance, an edge from an `#ifdef` node to a statement node represents that the statement is nested inside that `#ifdef` block. The C-preprocessor directive nodes are not integrated into the AST structure, that is, there are no AST edges to these nodes.

The authors do not explicitly describe how variability that changes the AST structure is handled. A discussion with the authors and an inspection of their tool implementation reveals that the the C-source code is parsed as if there were no C-preprocessor directives. The variability information is added as an addition to the traditional code property graph. This means that *undisciplined* C-preprocessor directives are not accurately handled by this approach. For instance, *undisciplined* C-preprocessor constructs can lead to changing the control flow depending on the variability. In this case, it would be required to annotate control flow edges in the code property graph with variability. However, this is not considered in the approach by Gerling et al.

Overall, variability-aware code property graphs offer a richer representation of source code, compared to the simple syntactical structure offered by the srcML-extractor. A richer representation can be helpful in the creation of code metrics. For instance, a metric for the control flow structure of source code could be implemented more easily when the control flow graph of the program is present in the code model. With the model created by the srcML-extractor, the metric first has to create the control flow structure based on the AST. However, the richer representation of the variability-aware code property graphs also requires more detailed parsing of the underlying code. At least with the current implementation of this approach, this leads to problems when parsing code with *undisciplined* conditional compilation directives.

Technically, Gerling et al. implemented their approach by writing the extracted code property graph to a graph database. Analyses of the code model are then implemented as queries to the graph database. For example, metrics could be implemented by queries that count certain elements in the graph. This poses a technical challenge to the integration in the KernelHaven architecture, which has a different approach for implementing analyses. In KernelHaven, the models are available as Java objects to the analyses, which are implemented in imperative Java code. Implementing analyses as queries on a graph databases cannot be integrated into this structure in a simple way. This means that the existing tooling developed by Gerling et al. can most likely not be re-used as-is. In contrast, for the srcML-extractor the srcML tool was re-used without modification, which saved a significant amount of development effort.

## 5 Evaluation

This chapter evaluates different components that were implemented in the context of this thesis. Each of the following sections contains the evaluation of a single component: Section 5.1 evaluates the non-Boolean transformation implemented for the Bosch PS-EC product line; Section 5.2 evaluates the `CodeBlockExtractor`, which parses C-preprocessor conditional compilation blocks for KernelHaven; Section 5.3 evaluates the approaches to simplify Boolean formulas that are applied in the Bosch PS-EC analysis; Section 5.4 evaluates the `srcML-extractor` that parses C-source code for KernelHaven.

The individual evaluation sections are all structured in the same way. After a short introduction of the component that is evaluated, the setup of the evaluation is described. This explains which parts of the component are evaluated and what measurements are conducted. After that, the results of the measurements are presented. Finally, the results of the evaluation and broader aspects of the evaluated component are discussed. The last section of this chapter, Section 5.5, will discuss the threats to validity of the evaluations. This is combined for all evaluations, as many of the arguments apply to more than one evaluation section.

### 5.1 Non-Boolean Transformation

This section evaluates the `NonBooleanPreperation` implementation for the KernelHaven infrastructure [KE18c]. This preprocessing components transforms the integer-based variability conditions of the PS-EC product line to propositional logic (cf. Section 3.2). Here, the conversion of the C-preprocessor conditions is evaluated, which is performed while copying the whole source code tree.

#### 5.1.1 Setup

The `NonBooleanPreperation` implementation is evaluated twice: in the context of a full execution of the KernelHaven pipeline and on generated artificial test cases. For the first evaluation, the full KernelHaven pipeline is run on two product variants of the PS-EC product line. This very closely reflects a realistic usage scenario. In this scenario, the `NonBooleanPreperation` will copy the source code tree and replace all integer-based C-preprocessor conditions with propositional ones. In the process, two aspects will be evaluated: the required *runtime*, and the *growth* of the propositional formulas compared to the original inter-based ones.

The measurement of the *runtime* will give a rough idea of the overhead that is required for the conversion from integer-based to propositional logic. The runtime measurements will be conducted by our industry partners at Bosch; we do not have control over the execution circumstances. The measurements will be done four times, and the median will be reported to remove some unwanted influences. The following individual timings will be measured for each of the product variants:

- The total runtime of the preprocessing component (**Measurement 1**). This includes all steps that are performed as part of the preparation (extracting the variability variable information, copying files, converting conditions).
- The total runtime spent on converting conditions (**Measurement 2**). The individual runtime of every single conversion of a C-preprocessor condition will be measured. This involves the three steps described in Section 3.2: parsing, transformation, and converting back to a C-preprocessor string. The sum of these individual measurements represents the total conversion time.
- The total runtime spent on copying source code files (**Measurement 3**). The individual runtime of every single source code file copy will be measured. The sum of these individual measurements represent the total time required for copying the source code files. Source code files are all files in the source code tree that end with `.c` or `.h`; copying of other files is not included in this measurements. This measurement also includes the conversion of C-preprocessor conditions, which is performed during the source file copying.

The *growth* of the condition sizes will be measured (**Measurement 4**). As explained in Section 3.2, an increase in the size of conditions is expected when converting from integer-based to propositional logic. This is because, generally, many Boolean variables are required to encode the same satisfiability properties as the original integer-based formula. The size of the conditions here will be simply measured as the number of characters in the C-preprocessor line. This includes the C-preprocessor directive at the beginning of the line (e.g. `#if`). Line continuations (via a `\` at the end of the line) will be appended together before measuring the length, so that the whole condition is captured. The length of the condition before and after the conversion are measured. The condition growth is calculated by dividing the length of the propositional condition by the length of the original, integer-based condition.

In addition to the growth and the runtime, the number of skipped conditions will be measured (**Measurement 5**). This will be conducted by counting the number of relevant exceptions that are present in the log file created by KernelHaven. The exceptions represent C-preprocessor lines that cannot be parsed, constructs in the AST that cannot be transformed, or other errors in the implementation (cf. Section 3.2 for a discussion of the limitations of the implementation).

In the second evaluation scenario, the `NonBooleanPreperation` is executed on artificially generated test cases [KES18a]. The test cases consist of 100 generated C-source code files which contain `#if` conditions. The conditions are generated in a way that all transformation rules described in Section 3.2 are covered. Five integer variability variables are used in the conditions, which all have a defined set of allowed values.

With these artificial test cases, the *runtime* of the non-Boolean transformation process is evaluated. Two different measurement series are conducted that measure how the runtime changes when generation parameters of the test cases are modified:

- The runtime of the `NonBooleanPreperation` with different numbers of `#if` conditions to transform will be measured (**Measurement 6**). For this, 20 test cases will be generated that have a varying number of `#if` conditions in each source code file. The number of conditions is increased in steps of 50, starting from 50 and up to 1000 conditions per file. With 100 files per generated test case, this results in a range of 5,000 to 100,000 total `#if` conditions. The set of allowed values for the 5 integer

variability variables is  $R(VAR) = \{1, 2, 3, 4\}$  and remains constant throughout all generated test cases.

- The runtime of the **NonBooleanPreperation** with different numbers of allowed values per integer variability variable will be measured (**Measurement 7**). For this, 17 test cases will be generated that have sets with varying sizes for the allowed values of variability variables. In a test case, all 5 variability variables have the same set of allowed values:  $R(VAR) = \{1, \dots, n\}$ , where  $n$  starts at 2 for the first generated test case and goes up to 18 for the last test case. The number of **#if** conditions stays constant, with 10 conditions in each of the 100 files.

The non-Boolean transformation has a fixed upper limit for the number combinations that are computed per integer operation (cf. Section 3.2). The measurements in this series will be conducted two times, once with the limit *enabled* and once with the limit *disabled*. The measurement with the limit *enabled* will show the runtime behavior that will occur in actual applications of the non-Boolean transformation, where this limit is enabled. The measurement with the limit *disabled* will show the runtime behavior of the underlying approach.

The execution time of the full **NonBooleanPreperation**, that is copying the generated source code files and converting all **#if** conditions, is measured for these two series. The measurements are conducted on a machine running Windows 7 with an Intel Core i7-6700 CPU with 8 cores @ 3.40 GHz and 16 GiB of RAM. For each of the generation settings described above, a test case is created and the **NonBooleanPreperation** is executed four times on it. The median runtime of these four measurements is then reported as the runtime for that setting.

### 5.1.2 Results

This section presents the results of the measurements described in the previous section. Table 5.1 shows the results for **Measurement 1** through **Measurement 5** for both PS-EC product variants. Note that, for **Measurement 4** and **Measurement 5**, the absolute number of C-preprocessor conditions cannot be reported due to confidentiality reasons. Thus, only the aggregate values shown here are available.

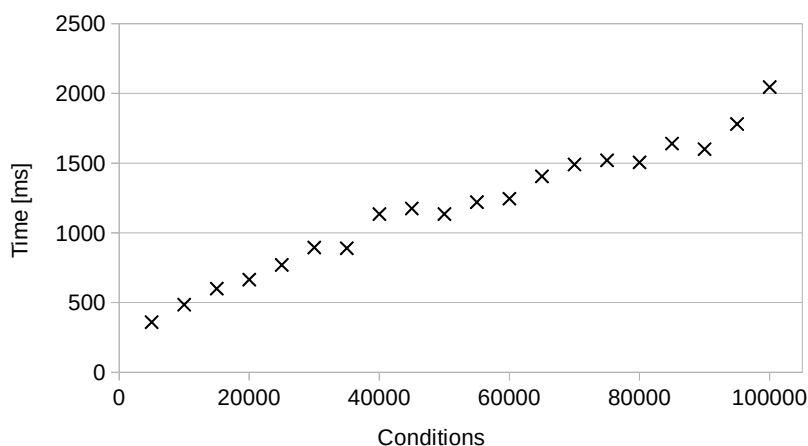


Figure 5.1: Runtime with varying number of conditions [KES18a]

Product Variant Measurement	Variant 1	Variant 2
<b>Measurement 1</b> (total preprocessing runtime)	49.9 s	56.1 s
<b>Measurement 2</b> (C-preprocessor conversion runtime)	0.6 s	0.7 s
<b>Measurement 3</b> (copying source code files runtime)	13.8 s	16.2 s
<b>Measurement 4</b> (condition size)	min: 14.71 % med: 121.21 % avg: 142.51 % max: 8675 %	min: 9.26 % med: 125.00 % avg: 150.98 % max: 8675 %
<b>Measurement 5</b> (unparseable conditions)	0.6 %	0.08 %

Table 5.1: Non-Boolean transformation measurements for two product variants.

Figure 5.1 shows the results of **Measurement 6**. The x-axis shows the total number of `#if` conditions in the generated test case and the y-axis shows the runtime of the `NonBooleanPreparation`. Each mark shows the median runtime of four executions for a generated test case.

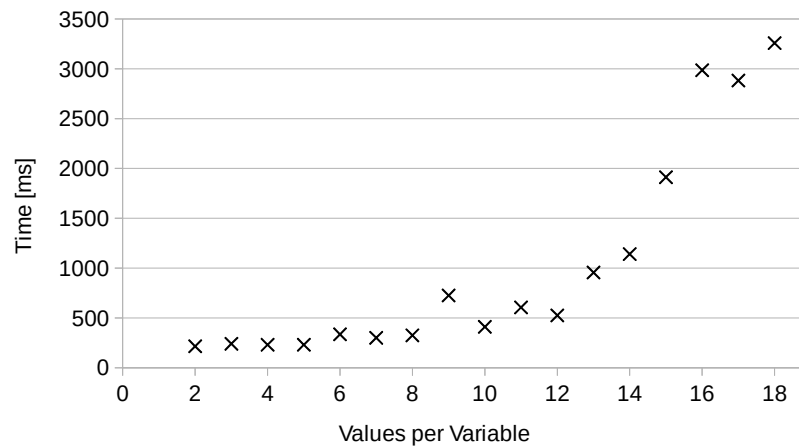


Figure 5.2: Runtime with varying number of allowed values [KES18a]

Figure 5.2 shows the results of **Measurement 7** with the combinatorial limit *disabled*. The x-axis shows the number of allowed values per integer variability variable and the y-axis shows the runtime of the `NonBooleanPreparation`. Each mark shows the median runtime of four executions for a generated test case. With the combinatorial limit *enabled*, the runtime for all test cases stays below 500 milliseconds.

### 5.1.3 Discussion

The `NonBooleanPreparation` runs as a preprocessing step to adapt the Bosch PS-EC product line to analyses that are defined for Boolean variability. Its runtime can be considered to be a pure *overhead* for the analysis; the transformations performed by it do

not help the analysis goal, but simply enable the analysis to be conducted at all. A fair assessment of the overhead would be to compare it to the effort of an analysis that directly handles integer-based variability itself. As explained in Section 3.2, an existing analysis would need to be technically adapted to handle the more powerful integer-based logic. This involves adapting the parser, data model (i.e. the abstract syntax tree required to represent the parsed conditions), and the SAT-solvers used on the conditions. The performance of such an adapted analysis could then be compared to the overhead of the non-Boolean transformation component.

However, Section 3.2 also explains another dimension where existing analysis approaches need to be adapted: the analysis need to be *conceptually* extended, to be able to handle integer-based variability. This is because they were previously defined on propositional logic only. This is especially a challenge for the feature-effect analysis, which has a basic assumption in its core definition that variability variables are Boolean. Comparing the non-Boolean transformation to the adaption of existing analyses would also need to consider this conceptual level, which cannot be measured like the technical runtime overhead. We argue that adapting an analysis to handle more than propositional logic likely involves more effort than the non-Boolean transformation approach that was developed in this thesis. The non-Boolean transformation also has the benefit that it allows *all* existing SAT-based analyses to be executed on the Bosch PS-EC product line. Adapting *all* these analyses to integer-based variability would require even more effort.

The non-Boolean transformation also has a few disadvantages compared to analyses that directly support integer-based variability. The non-Boolean transformation can only be applied on product lines where the integer variability variables mostly have only a small set of allowed values. While this is true for the Bosch PS-EC product line, this is not necessarily the case for all product lines with integer-based variability. Additionally, the non-Boolean transformation has cases where it cannot transform a condition exactly, because it contains unrestricted integer variables (cf. Section 3.2). In this case, an analysis that can handle integer-based variability directly may be able to more accurately handle these cases. Also the analyses may need to be slightly extended, as the non-Boolean transformation has implicit constraints on the introduced artificial Boolean variables. For the feature-effect analysis, for instance, this involves additional steps that set some variables to *false* and aggregate feature-effects together (cf. Section 3.4).

Due to the lack of an adapted analysis that can directly handle integer-based variability, the overhead of the non-Boolean transformation cannot be compared to such an analysis. Instead, the overhead is compared to the overall runtime of the non-adapted analysis (in this evaluation, this is the feature-effect analysis presented in Section 3.4). In the Bosch PS-EC use-case, the runtime heavily depends on which simplification strategy is selected. With the fastest simplification approach (the “prune constants” approach from Section 3.5) the complete runtime of the KernelHaven pipeline is 57 seconds for the first and 1 minute 42 seconds for the second measured product variant. In this scenario, the 49.9 seconds and 56.1 seconds runtime (**Measurement 1**) of the `NonBooleanPreparation` make up about 87.5 % and 55.0 % of the runtime. However, as discussed in Section 3.5, this simplification strategy creates inadequate results. The next simplification level that can be configured (enabling the simplifying disjunction) increases the runtime of the KernelHaven pipeline to 51 minutes and 3 seconds for the first and 3 minutes and 26 seconds for the second measured product variant. In this scenario, the `NonBooleanPreparation` only makes up 1.2 % of the runtime for the first and 24.8 % of the runtime for the second product variant. For the first product variant, this can be considered to be insignificant. For the

second product variant, this already shows a significant reduction in the relative runtime of the preparation. Further simplification levels decrease this even further. Additionally, our experience with other SAT-based analyses of product lines in the past showed that an added overhead of under a minute is usually not significant.

The next two measurements give insight on the runtime of individual steps inside the preprocessing. **Measurement 3** shows that the copying of source code files takes 13.8 and 16.2 seconds for the two product variants, which leaves 36.1 and 39.9 seconds of the complete preprocessing runtime (**Measurement 1**) for other tasks. These other tasks are copying non-source code files and waiting for the variability model to be extracted. The `NonBooleanPreparation` requires the variability model as it needs the set of allowed integer values for the transformation of variability variables.

**Measurement 2** shows that the actual conversion of C-preprocessor conditions takes only 0.6 or 0.7 seconds (depending on the product variant), that is only about 4.3 % of the runtime required for copying source code files (**Measurement 3**). This comparison shows that a large overhead comes from the strategy to copy the source tree completely and then running the source code extractor on that copy. This strategy was chosen as it opaquely preprocesses the source code files for the extractor and does not modify the original files. Different strategies here could be implemented that reduce the overhead of the non-Boolean transformation. For example, the C-preprocessor conditions could be replaced in the original source code files, instead of copying the whole file. Alternatively, the non-Boolean transformation could be implemented in the source code extractor. This would then remove the need for a preprocessing component that modifies files; the transformation could be done on-the-fly in-memory, while the conditions are read from the source code files.

**Measurement 4** compares the size of the original, integer-based C-preprocessor condition to the propositional replacement. In most cases, the formulas grow a bit; in the first measured product variant the median is 121.21 % and the average is 142.51 %, which signify a growth rate of 21.21 % and 42.51 % respectively. For the second measured product variant, the median growth rate is 25.00 % and the average growth rate is 50.98 %. These are slightly higher than the ones for the first product variant, but remain in the same magnitude. In some cases, the formula even shrinks in size, with a reduction of 90.74 % marking the extreme case in the second product variant. This may be due to legacy variability being pruned and unsatisfiable parts of the condition, which are detected during the transformation, being dropped or replaced by constants. On the other side, some conditions grow heavily, with a growth of more than 85 times. However, the lower median compared to the average indicates that the bulk of formulas is skewed towards smaller growth, with fewer bad cases increasing the average. Overall, these growth rates indicate that the general approach of Boolean replacements for integer-based conditions is feasible. A few very high growth rates create extreme cases, but the PS-EC use-case shows that these large formulas can be handled by the following feature-effect analysis.

**Measurement 5** measures how many C-preprocessor conditions cannot be parsed or transformed by the `NonBooleanPreparation` implementation. We consider a ratio of 0.6 % of unparseable conditions (or even 0.08 % for the second product variant) to be acceptable, considering the quirks of the C-preprocessor (cf. Section 3.2). The parsing and transformation issues do not stem from conceptual problems with the non-Boolean transformation approach, but rather they are problems of the specific implementation. With more engineering effort, the implementation could be extended to correctly handle all corner cases of the C-preprocessor and transformation approach.

**Measurement 6** shows that the non-Boolean transformation approach scales linearly with the number of `#if` conditions in the product line [KES18a]. This is because each condition is transformed independently of the others. For a full product line, this means that the approach scales linearly with the size of the product line, as generally the number of variability conditions increases with the size of the product line. The complexity of the individual variability conditions, which would increase the runtime of the transformation approach, does not necessarily increase with the size of the product line. Thus, the approach is also applicable for large-scale software product lines. This was also confirmed in the Bosch PS-EC use-case, where the size of the product variants that were analyzed did not cause runtime problems for the non-Boolean transformation.

Finally, **Measurement 7** shows how the non-Boolean transformation approach scales when the integer variability variables have larger sets of allowed values [KES18a]. When an integer operator has variability variables on both sides, the transformation approach considers all combination of the allowed values for both variables. This leads to a quadratic growth of combinations that need to be considered, which can be seen in the increase of required runtime in Figure 5.2. The underlying idea of the non-Boolean transformation approach is to introduce Boolean variables for each allowed value of the integer variables. This is only feasible if the majority of variability variables only have a *small* set of allowed values. For **Measurement 7**, test cases were generated where *all* variability variables have larger amounts of allowed values. The measured runtime supports that the approach is only feasible for smaller sets of allowed values.

Figure 5.2 of **Measurement 7** shows the runtime with the combinatorial limit of the transformation approach *disabled*. If it is *enabled*, the implementation has a defined cut-off point, where integer operators that need to compare more than this amount of combinations will not be evaluated exactly. **Measurement 7** showed that if the limit is *enabled*, the runtime of all the generated test cases stays below 500 milliseconds. The limit helps to mitigate the performance problems of larger sets of allowed values, but it comes at the cost of a less exact transformation (cf. Section 3.2). The test cases for **Measurement 7** are generated in a way that almost all `#if` conditions trigger cases where many combinations need to be considered. This is because all variability have the same (large) set of allowed values.

In practice, the majority of variability variables in the PS-EC product line have a small set of allowed values. Only some variability variables have larger sets of allowed values and they mostly do not appear in the same `#if` condition. Because of this, only a very small amount of `#if` conditions can cause performance problems in the non-Boolean transformation. The combinatorial limit mitigates the runtime problems of these few cases by using a less exact transformation. However, the vast majority of conditions can be transformed with the default exact approach.

## 5.2 CodeBlockExtractor

This section evaluates the CodeBlockExtractor implementation in KernelHaven [Kra18], which is introduced in Section 3.3. The CodeBlockExtractor reads the C-source code files of a software product line and builds a model representing the variability conditions in the conditional compilation blocks of the C-preprocessor. It supports variability conditions in propositional logic, with Boolean variability variables implemented by defining or not defining a C-preprocessor symbol.

### 5.2.1 Setup

The CodeBlockExtractor is evaluated on the Linux Kernel, a large open-source software product line [Lin19]. Using an open-source product line as the basis for evaluation allows for transparent reporting of measurements and makes the measured results reproducible by others. The variability in the Bosch PS-EC product line (cf. Section 3.1) is transformed to the variability implementation used in the Linux Kernel (cf. Section 3.2). Thus, the insights gained by evaluation the CodeBlockExtractor on the Linux Kernel are applicable for the Bosch PS-EC use-case.

The latest Linux Kernel version as of writing, version 5.3.5, will be used to evaluate the CodeBlockExtractor. As the Linux Kernel tends to grow in size with each new release, this likely represents the version with the most files and C-preprocessor conditions to parse. The CodeBlockExtractor is configured to parse all files ending in `.c` or `.h`. The total number of these files will be measured (**Measurement 1**). In addition, the total number of conditions in the C-preprocessor blocks will be measured (**Measurement 2**). This measurement is conducted on the parsing results of the CodeBlockExtractor. Thus, files that could not be parsed are not included in this measurement. The CodeBlockExtractor can be configured to insert artificial top-blocks for source code files that contain code outside of all C-preprocessor blocks. This influences the number of conditions that the extractor creates. Thus, **Measurement 2** will be reported for both possible settings. Further measurements will be conducted to assess the *performance* and the *robustness* of the parser.

For the *performance*, the total runtime required to extract all source code files will be measured (**Measurement 3**). This measurement is performed in a virtual machine running Ubuntu 16.04 with 40 logical CPU cores of an Intel Xeon E5-2650V3 @ 2.3 GHz and 314 GiB RAM. The extractor is configured to only use a single execution thread. The extraction will be run once as warmup, and then be repeated 4 times.

The performance of the CodeBlockExtractor will be compared to the UndertakerExtractor. The UndertakerExtractor is another code model extractor in KernelHaven [KE17]. It has similar capabilities to the CodeBlockExtractor, as it also represents the C-preprocessor variability of source code files as a hierarchy of conditional compilation blocks with presence conditions. It uses the parser of the Undertaker tool [TLS+11] to extract the block information from the source code files. **Measurement 3** will also be conducted for the UndertakerExtractor: the setting is exactly the same, except the UndertakerExtractor instead of the CodeBlockExtractor is used as the code extractor in KernelHaven.

To assess the *robustness* of the CodeBlockExtractor, two measurements will be done to check where the parser cannot parse something. The CodeBlockExtractor may fail to parse a file; this can be due to an incorrect nesting structure of C-preprocessor blocks, or due to bugs in the implementation of the extractor. The number of files that cannot be parsed will be measured (**Measurement 4**). The CodeBlockExtractor can also fail to parse conditions of C-preprocessor blocks. To measure the amount of unparseable conditions, the extractor is configured to insert a special artificial variable in place of the unparseable conditions. In the parsing results, the number of occurrences of this variable will be counted (**Measurement 5**). A special parsing mode called *fuzzy parsing* can be enabled for the CodeBlockExtractor, which can handle integer comparisons in the C-preprocessor conditions (cf. Section 3.3). **Measurement 5** will be conducted twice, once with the mode enabled and once with it disabled.

### 5.2.2 Results

The results for the measurements on Linux 5.3.5 are as follows: the total number of `.c` and `.h` files in the Linux Kernel is 47,749 (**Measurement 1**). 0 files could not be parsed by the extractor (**Measurement 4**). The total number of C-preprocessor conditions created by the extractor, including artificial top-blocks, is 129,985 (**Measurement 2**). With artificial top-blocks disabled, the total number of C-preprocessor conditions is 100,999; thus 28,986 artificial top-blocks are created if that setting is enabled. The total number of C-preprocessor conditions includes the conditions that could not be parsed: 5,132 conditions (about 5.1 % of the total, excluding top-blocks) were replaced by an error variable because they could not be parsed (**Measurement 5**). With *fuzzy parsing* enabled, the number of unparseable conditions drops to 753 (about 0.7 % of the total).

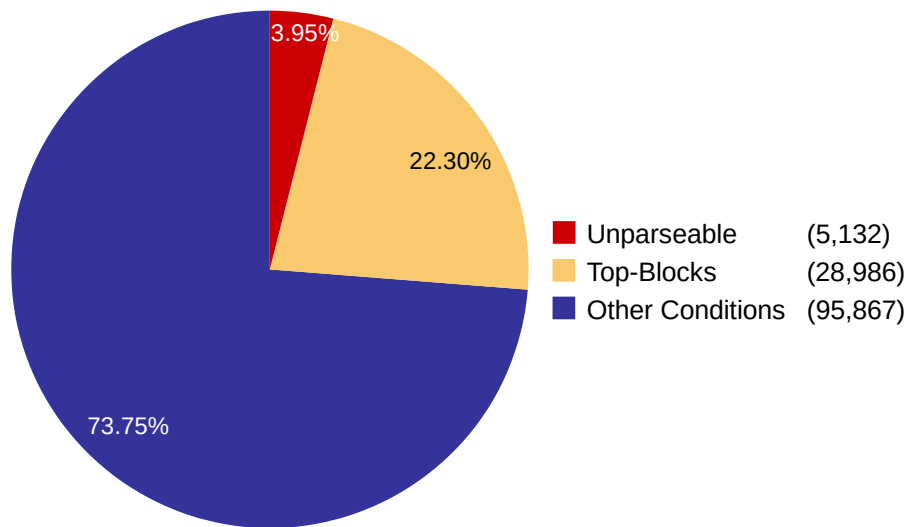


Figure 5.3: Ratio of extracted condition types

Figure 5.3 compares the amounts of different types of conditions. The total amount of conditions, used as the 100 % baseline, is all conditions extracted by the CodeBlock-Extractor, including the artificial top-blocks (**Measurement 2**). Of these, the relative amount of unparseable (**Measurement 5** without *fuzzy parsing*) conditions (3.95 %) and artificial top-blocks (22.30 %) is shown. This leaves 73.75 % of the conditions that are parseable and extracted from the source code files (i.e. they are not artificially created).

**Measurement 3** was performed four times, with a preceding warmup execution. The individual runtimes of the CodeBlockExtractor on a single thread were: 10.133 s, 10.375 s, 10.793 s, and 10.070 s. This results in a median execution time of 10.254 seconds. On average, each of the 47,749 files (**Measurement 1**) was parsed in about 0.215 milliseconds.

**Measurement 3** was also performed for the UndertakerExtractor. The individual runtimes on a single thread were: 11 minutes 30.006 seconds, 11 minutes 30.225 seconds, 11 minutes 30.396 seconds, and 11 minutes 30.107 seconds. This results in a median execution time of 11 minutes 30.166 seconds. On average, each of the 47,749 files (**Measurement 1**) was parsed in about 14.454 milliseconds.

### 5.2.3 Discussion

The CodeBlockExtractor parses C-preprocessor conditional compilation blocks in source code files to a hierarchical structure. For this, the conditions in the C-preprocessor blocks are parsed as Boolean formulas. They are combined to form the presence condition of each block (cf. Section 3.3). One of the main concerns of the extractor is to create an accurate representation of the variability in the source code. The correctness of the hierarchical structure and computation of presence conditions is tested by a test suite that was created during the development of the extractor. This test suite also tests the parser for the C-preprocessor conditions. However, in real-world product lines like the Linux Kernel, there is a wide variety in the C-preprocessor condition usage. The C-preprocessor allows many different constructs in `#if` conditions, many of which cannot be parsed as Boolean operations (e.g. number arithmetic). Compared to that, the correct creation of the block hierarchy and computation of presence conditions is relatively easy. Thus, a main concern of the conducted measurements is to evaluate how many conditions can be correctly parsed by the extractor.

The CodeBlockExtractor was implemented to raise exceptions in all cases where it does not detect that it can correctly parse a given construct. For example, if unknown operators or macro calls are detected in a condition, an exception is created. The extractor does not try to parse constructs where it already partially failed. Because of this, the evaluation uses the exceptions created by the extractor as a basis to assess how many constructs cannot be parsed. **Measurement 4** showed that all files of the Linux Kernel could be parsed. This means that there were no parsing errors outside of unparseable C-preprocessor conditions. The unparseable conditions are replaced by special error variable; all other errors during parsing cause the parsing of the whole file to be aborted.

**Measurement 5** showed that 5.1 % of all C-preprocessor conditions in the source code files (**Measurement 2**, excluding artificial top-blocks) cannot be parsed. The ratio of unparseable conditions drops to 0.7 % when *fuzzy parsing* is enabled, which can handle some cases of numeric comparison operators. This shows that at least 85.3 % of the unparseable conditions are due to numeric operations in the C-preprocessor condition. Manual inspection of the remaining unparseable conditions reveals that the vast majority of the still unparseable conditions fall into one of two categories:

- Numeric operations that cannot be handled by *fuzzy parsing*. *Fuzzy parsing* can only handle some cases of numeric comparison, e.g. when a variable is compared to a literal (cf. Section 3.3). However, C-preprocessor conditions can also contain arithmetic operators. For instance, a condition that cannot be handled by *fuzzy parsing* is:

```
#if (IDP_CPLD_VIRT + IDP_CPLD_SIZE) > 0xfc000000
```

Such a condition with numeric operations cannot be directly parsed as a Boolean formula. Parsing this is out-of-scope for the CodeBlockExtractor, which is only meant to support Boolean variability.

- Custom macro calls in conditions cannot be handled. The C-preprocessor allows to define macros via the `#define` directive. Macros are functions which can be used in `#if` conditions. For example, a condition calling a macro function is:

```
#if OCTEON_IS_MODEL(OCTEON_CN38XX)
```

Support for parsing custom macros to Boolean formulas would require to evaluate previous `#define` directives. These can occur in header files that are included via `#include` statements. Additionally, there may be alternative definitions of macros in different conditional compilation blocks. Accurate evaluation of macros from `#define` directives quickly results in a very high complexity. Kästner et al. propose a variability-aware lexer, which can handle C-preprocessor macros in a software product line [KGR+11]. However, for the relatively simple CodeBlockExtractor, supporting custom macros is out-of-scope. It only has hard-coded support for three macros that are used throughout the Linux Kernel (cf. Section 3.3).

The CodeBlockExtractor was developed with the goal to replace the UndertakerExtractor. The UndertakerExtractor is based on the parser of the Undertaker tool [TLS+11]. It has similar capabilities, as both extractors parse the C-preprocessor blocks into a hierarchy with presence conditions. Two aspects motivated the creation of a successor for the UndertakerExtractor: *portability* and *performance*.

The Undertaker tool is developed in C++. To integrate it into the KernelHaven pipeline, which is written in Java, the UndertakerExtractor starts a new process for each source code file that is parsed. Undertaker can only be compiled and executed on a Linux machine; it does not support other platforms. Thus, the whole UndertakerExtractor, which starts an Undertaker parser process internally, can only run on Linux machines. KernelHaven itself is platform independent; it can run on all platforms that can run a Java virtual machine. In contrast to the limited UndertakerExtractor, the CodeBlockExtractor is fully implemented in Java. Thus, it supports all platforms that KernelHaven runs on.

The performance of the CodeBlockExtractor (0.215 milliseconds per file, **Measurement 3**) is more than 67 times higher than the performance of the UndertakerExtractor (14.454 milliseconds per file, **Measurement 3**). A part of that difference stems from the additional overhead the UndertakerExtractor has: it starts a new parser process for each source code file. The results of the parsing are serialized as text and sent over the standard output of the parsing process to the Java implementation. There, the result is deserialized and converted into the data model of KernelHaven. In contrast, the CodeBlockExtractor does not need to start any process and can directly use the data model of KernelHaven to represent the parsing results.

## 5.3 Formula Simplification

This section evaluates the Boolean formula simplification that is performed as part of the feature-effect analysis of the Bosch PS-EC product line. Section 3.5 introduced different approaches for simplifying formulas. Some of them are general (i.e. they can simplify any Boolean formula that is given to them), while others are specific to the feature-effect computation. This section evaluates all the approaches in the context of the feature-effect computation.

### 5.3.1 Setup

The Boolean formula simplification will be evaluated in the context of a full execution of the KernelHaven pipeline on two product variants of the PS-EC product line. This very

closely reflects a realistic usage scenario. In this scenario, the simplification will be used throughout the feature-effect computation process, as shown in Figure 3.9.

Four different simplification settings can be set in the KernelHaven pipeline configuration file. Each of them uses one or more of the simplification approaches that were presented in Section 3.5. The *runtime* and resulting *formula sizes* will be measured for each of the settings. For this, a the full KernelHaven analysis pipeline with a feature-effect analysis will be executed per simplification setting on each of the two PS-EC product variants. The four settings are:

**Simple** This setting only enables the “prune constants” simplifications for the XOR terms and the final simplification of the feature-effect. This is the most basic form of simplification. A setting which disables all simplifications does not exist, as such a configuration is nonsensical in the context of feature-effect computations (cf. the introductory example of Section 3.5). Thus, this lowest possible simplification setting will serve as a baseline to compare the other, more sophisticated approaches. This setting is achieved by setting

```
analysis.simplify_conditions = NO_SIMPLIFICATION and
logic.simplifier = SIMPLE in the KernelHaven configuration.
```

**Simplifying Disjunction** This setting uses the “prune constants” approach for all simplification steps in the computation process (i.e. also presence conditions are simplified with it). The setting additionally enables the simplifying disjunction strategy. This way, all simplification steps shown in Figure 3.9 are enabled in their most simple form. This setting is achieved by setting

```
analysis.simplify_conditions = PRESENCE_CONDITIONS and
logic.simplifier = SIMPLE in the KernelHaven configuration.
```

**Visitor** This setting basically equals the previous one, but uses the more sophisticated “simple Boolean algebra” approach for general formula simplification, instead of the “prune constants” one. This setting is achieved by setting

```
analysis.simplify_conditions = PRESENCE_CONDITIONS and
logic.simplifier = VISITOR in the KernelHaven configuration.
```

**AAS** This setting uses the “combined approach”, which internally makes use of all other general simplification approaches. It also has the simplifying disjunction enabled, like the two previous settings. This is the most sophisticated simplification setting possible in KernelHaven. This setting is achieved by setting

```
analysis.simplify_conditions = PRESENCE_CONDITIONS and
logic.simplifier = ADAMS_AWESOME_SIMPLIFIER in the KernelHaven configuration.
```

The *runtime* of the feature-effect computation will be measured (**Measurement 1**), which contains all simplification steps except the preliminary presence condition simplification. The number of feature-effects that are computed cannot be reported due to confidentiality reasons. Thus, only the total runtime of this step can be disclosed. The runtime of the simplification of presence conditions is measured separately (**Measurement 2**), as it is implemented in a different analysis component than the feature-effect. In contrast to these measurements, which are collected for all settings, the following measurements are only collected for some of the settings that they are applicable for:

- The total runtime spent in the general formula simplification is measured for the *Visitor* and *AAS* settings (**Measurement 3**). This involves all “simplification” steps

shown in Figure 3.9, that is the simplification of presence conditions, XOR trees, and the resulting feature-effect formula. Additionally, the simplification after the aggregation of feature-effects (cf. Section 3.4) is also included.

- The runtime spent in the simplifying disjunction is measured for the settings that have it enabled (i.e. the *Simplifying Disjunction*, *Visitor*, and *AAS* settings). For this, the total runtime spent in the simplifying disjunction (**Measurement 4**) and the individual runtimes for each term that is added to a simplifying disjunction (**Measurement 5**) are measured.
- For the *AAS* setting, i.e. the “combined approach”, the total runtime spent on each of the individual steps (cf. Section 3.5) is measured (seven steps, **Measurement 6** through **Measurement 12**). Additionally, the runtime of the individual steps of the “equal sub-formula detection” approach, which is used as a part of the “combined approach”, are measured. These steps are:
  1. Searching for equal sub-formulas (**Measurement 13**)
  2. Replacing of equal sub-trees with a temporary variable (**Measurement 14**)
  3. Simplifying with the “more complex Boolean algebra” approach (**Measurement 15**)
  4. Checking if the formula has changed during simplification (**Measurement 16**)
  5. Replacing of the temporary with the original sub-formula (**Measurement 17**)

The runtime measurements will be conducted by our industry partners at Bosch; we do not have control over the execution circumstances. The feature-effect implementation is parallelized over multiple execution threads. The runtime measurements are conducted on each thread and summed up to get the total runtime. For example, 10 threads running 1 minute each would result in a total measured runtime of 10 minutes.

In addition to the runtime, the resulting feature-effect formula sizes will be measured for each of the four settings. This will allow a comparison of how much the different approaches can simplify formulas. The main goal of the simplification is to create *understandable* formulas, but as explained in Section 3.5 this can be reduced to the problem of formula *size*. Due to confidentiality reasons, no absolute numbers for the size or number of variability variables can be given. Instead, all measurements will be given as ratios. The following measurements will be conducted on the final feature-effect formulas that are created by the analysis:

- The number of feature-effects that are *true* (i.e. top-level/independent variables) will be measured (**Measurement 18**).
- The total number of Boolean operators in all feature-effects will be counted (**Measurement 19**).
- The total number of variables in all feature-effects will be counted (**Measurement 20**)

### 5.3.2 Results

<b>Setting</b> <b>Measurement</b>	Simple	Simplifying Disjunction	Visitor	AAS
<b>Measurement 1</b> (total feature-effect runtime)	44.095 s	2:18:17.244	2:17:52.525	37:28:58.372
<b>Measurement 2</b> (presence condition simplification)	0 s	0.822 s	0.822 s	17:17.913
<b>Measurement 3</b> (total general formula simplification runtime)	—	—	5.058 s	37:12:19.292
<b>Measurement 4</b> (total simplifying disjunction runtime)	—	2:18:13.285	2:17:46.722	1:48:44.652
<b>Measurement 5</b> (simplifying disjunction, individual terms)	—	med: 0.003 s avg: 0.082 s max: 8.880 s	med: 0.002 s avg: 0.083 s max: 9.595 s	med: 0.001 s avg: 0.063 s max: 7.675 s

Table 5.2: Runtime measurements of different simplification settings (variant 1)

<b>Setting</b> <b>Measurement</b>	Simple	Simplifying Disjunction	Visitor	AAS
<b>Measurement 1</b> (total feature-effect runtime)	1:12.854	8:57.950	11:06.876	53:23.414
<b>Measurement 2</b> (presence condition simplification)	0 s	0.581 s	1.221 s	15:57.728
<b>Measurement 3</b> (total general formula simplification runtime)	—	—	6.159 s	1:15:28.198
<b>Measurement 4</b> (total simplifying disjunction runtime)	—	8:53.130	10:58.630	3:29.422
<b>Measurement 5</b> (simplifying disjunction, individual terms)	—	med: 0.001 s avg: 0.004 s max: 0.637 s	med: 0.001 s avg: 0.005 s max: 0.997 s	med: 0.000 s avg: 0.001 s max: 7.693 s

Table 5.3: Runtime measurements of different simplification settings (variant 2)

Table 5.2 shows the results for the runtime measurements **Measurement 1** through **Measurement 5** for the first product variant. Table 5.3 shows the results for the second product variant. The measurements are presented in different formats, depending on their magnitude. If they are under a minute, the timings are formatted as seconds, with millisecond precision as decimals. If they are under an hour, they are formatted as *minutes:seconds.milliseconds*. The measurements greater than one hour are formatted as *hours:minutes:seconds.milliseconds*.

<b>Product Variant</b> <b>Measurement</b>	Variant 1	Variant 2
<b>Measurement 6</b> (Step 1 of “combined approach”, simplify with “prune constants” approach)	2.276 s	2.573 s
<b>Measurement 7</b> (Step 2 of “combined approach”, move negations inwards)	0.774 s	0.954 s
<b>Measurement 8</b> (Step 3 of “combined approach”, simplify with “simple Boolean algebra” approach)	3.573 s	3.992 s
<b>Measurement 9</b> (Step 4 of “combined approach”, simplify with “more complex Boolean algebra” approach)	22.852 s	26.670 s
<b>Measurement 10</b> (Step 5 of “combined approach”, simplify with “equal sub-formula” approach)	20:11:44.942	1:02:36.921
<b>Measurement 11</b> (Step 6 of “combined approach”, move negations outwards)	5.045 s	5.443 s
<b>Measurement 12</b> (Step 7 of “combined approach”, simplify with “equal sub-formula” approach)	16:59:59.830	12:11.645
<b>Measurement 13</b> (Step 1 of “equal sub-formula” approach, search equal sub-formulas)	56.304 s	46.256 s
<b>Measurement 14</b> (Step 2 of “equal sub-formula” approach, replace with temporary variable)	13:45.539	4:10.686
<b>Measurement 15</b> (Step 3 of “equal sub-formula” approach, simplify with “more complex Boolean algebra” approach)	57:35.966	22:07.272
<b>Measurement 17</b> (Step 4 of “equal sub-formula” approach, check if formula has changed during simplification)	35:59:09.887	47:31.691
<b>Measurement 17</b> (Step 5 of “equal sub-formula” approach, replace temporary variable with original sub-formula)	2.989 s	2.609 s

Table 5.4: AAS setting-specific runtime measurements (both variants)

The *AAS* setting-specific runtime measurements for both product variants are shown in Table 5.4. For the first product variant, the total time spent in the “combined approach” simplification is about 37 hours and 12 minutes (**Measurement 3**). 99.97 % of that is spent on steps 5 and 7 (**Measurement 10** and **Measurement 12**). These two steps are the “equal sub-formula detection” simplification approach. For the second product variant, the relative time spent in the “equal sub-formula detection” approach is 99.12 %.

The time spent in the “equal sub-formula detection” approach for the first product variant is 37 hours, 11 minutes, and 44.772 seconds. 96.76 % of that is spent in step 4 (**Measurement 17**). For the second product variant, the relative time spent in step 4 of the “equal sub-formula detection” approach is only 63.67 %.

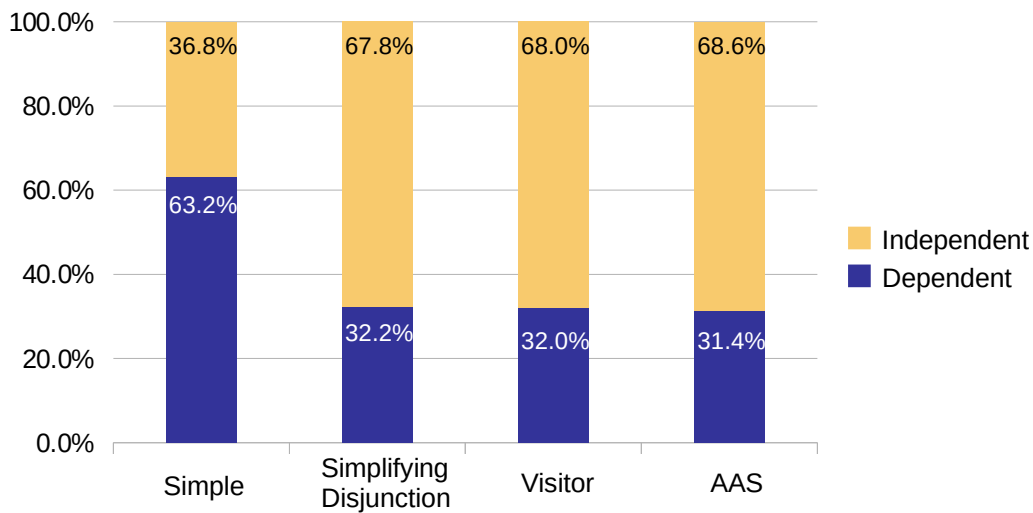


Figure 5.4: Ratio of dependent vs. independent variables for different simplifications

Figure 5.4 shows the results of **Measurement 18** for the first product variant. For each simplification setting, the relative amount of dependent and independent variability variables is shown. The absolute number of variability variables is equal for all simplification settings. The number of independent variables for the *Simple* setting (about 37 %) is lower than the number of independent variables for the other settings (about 68 %). The results for **Measurement 18** for the second product variant show a similar growth in the number of independent variables: about 17 % are independent with the *Simple* setting, while about 36 % are independent with the other settings.

Figure 5.5 shows the results of **Measurement 19** and **Measurement 20** for the first product variant. The number of operators and variables in all resulting feature-effect formulas were measured. The results for the *Simple* setting are used as baseline. The results for the other settings are shown in Figure 5.5 as relative percentages of that baseline. The *Simple* setting would have values of 100 % for both, the number of operators and variables; it is not shown in this diagram. The settings *Simplifying Disjunction* and *Visitor* have similar results, with about 4.1 % the number of operators and 3 % the number of variables compared to the *Simple* setting. The *AAS* setting reduces these values further to 1.57 % (about 60 % improvement over the *Visitor*) and 1.34 % (about 55 % improvement over the *Visitor*) respectively.

For the second product variant, **Measurement 19** and **Measurement 20** show similar relations between the different simplification settings. The *Simplifying Disjunction* and

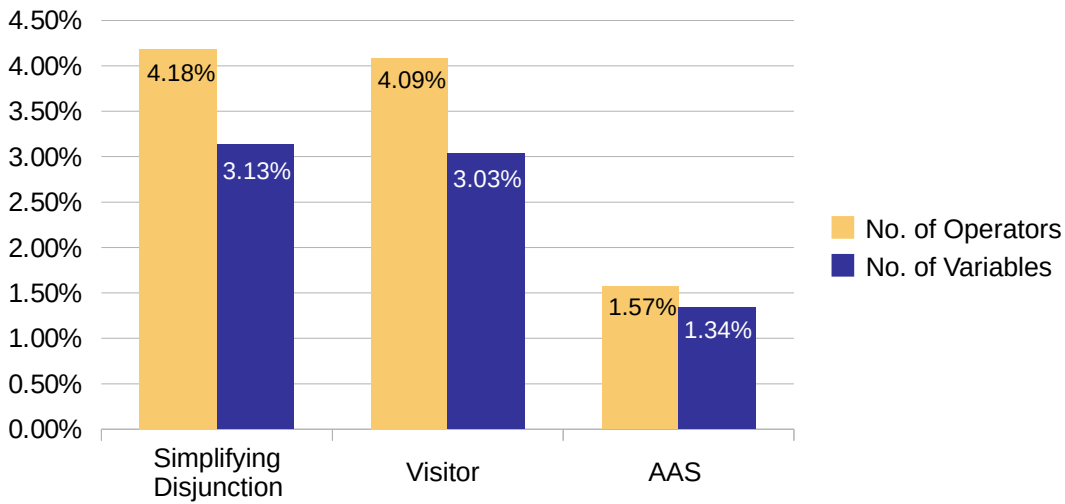


Figure 5.5: Total number of operators and variables relative to Simple simplification

*Visitor* settings drastically reduce the size compared to the *Simple* setting. The *AAS* setting has further improvements over the *Visitor* setting of about 70 % for both, the number of variables and operators.

### 5.3.3 Discussion

The simplification of Boolean formulas was introduced to counteract the tendency of the feature-effect approach to create large formulas. The feature-effect computation inherently creates formulas that contain many *true* and *false* constants (cf. Section 3.4). Thus, the “prune constants” approach was introduced to simplify the resulting feature-effect formulas. It uses the Boolean algebra rules *identity* and *annihilator* to remove all constants and simplify the formula a bit.

However, for the Bosch PS-EC use-case the formulas were still too large. The goal is to create concise feature-effect formulas that serve as a basis for domain experts to create variability model constraints. The formulas thus have to be *understandable* by humans, which for the case of Boolean formulas boils down to the *size* of the formulas (cf. Section 3.5). The feature-effects simplified with the “prune constants” approach were still too large for our industry partners at Bosch. Thus, further simplification strategies were developed.

Finding the smallest possible equivalent Boolean formula for a given feature-effect is not feasible for the number of variables that commonly occur in the feature-effect formulas. As Section 3.5 discussed, the exact Quine-McCluskey approach [Qui52; Qui55; McC56] to find the smallest equivalent is NP-complete. Common heuristics that reduce the runtime behavior, like the Espresso logic minimizer [BHH+82], still require the full truth-table as an input. Creating this for the arbitrary Boolean feature-effects is unfeasible, as it grows exponentially in size. We found that existing research tends to focus on formulas with few unique variables or on specialized cases (e.g. read-once Boolean functions, where each variable only occurs once). Thus, we developed our own simplification heuristics, which are based on Boolean algebra rules.

The “simplifying disjunction” is a specialized simplification approach, which is tailored to the specifics of the feature-effect definition (cf. Section 3.5). The core idea is to detect which terms of the main disjunction in the feature-effect definition (Formula 3.2) can be left out. **Measurement 19** and **Measurement 20** show that the *Simplifying Disjunction* setting greatly reduces the formula sizes, compared to the *Simple* setting. In addition, **Measurement 18** shows that this approach also leads to about twice as much feature-effects being simplified to *true*. These are then classified as independent/top-level variables. The detection of independent variability variables is a main concern of the feature-effect analysis (cf. Section 3.6), so increasing the detection rate is a benefit of the simplification approach.

The results of the *Simplifying Disjunction* setting (and the *Visitor* setting, which has similar results) can be further improved with the *AAS* setting. The *AAS* setting enables the “combined approach”, which internally utilizes all other simplification approaches. It is the only setting where the “equal sub-formula detection” approach is used, which increases the detection rate of Boolean algebra transformation rules that can be applied. While **Measurement 18** only shows a marginal increase of the feature-effects that are simplified to *true*, **Measurement 19** and **Measurement 20** show a significant improvement in the formula sizes. In the first product variant of the evaluation, the *AAS* setting lead to a further reduction of 55 % to 60 %. In the second product variant, the reduction is even higher at 70 %.

However, all simplification approaches have a significantly increased runtime. **Measurement 1** shows the runtime of the feature-effect computation, which includes all simplifications shown in Figure 3.9 except the preliminary presence condition simplification. For the first product variant of the evaluation, the runtime of the feature-effect computation for the *Simplifying Disjunction* setting is about 187 times longer than with the *Simple* setting. The runtime with the *AAS* setting is more than 3,000 times longer (15 times longer than with the *Simplifying Disjunction* setting). For the second product variant in the evaluation, the increase is much lower but still significant: the runtime of the feature-effect computation for the *Simplifying Disjunction* setting is about 6 times longer than with the *Simple* setting. The runtime with the *AAS* setting is about 43 times longer (5 times longer than with the *Simplifying Disjunction* setting).

The benefit of the smaller feature-effect sizes has to be weighed against the increase in computation runtime. If the goal to create smaller formulas is more important than the runtime, the most powerful simplification approach should be configured. This is for example the case in the Bosch PS-EC use-case: the product variants are analyzed once to create feature-effects that serve as a basis for domain experts to create variability model constraints. Here, the most important goal is that the constraints can be used by humans. The analysis is only conducted once, thus a longer runtime is tolerable. However, a different context may have different priorities. For instance, the feature-effect analysis could be conducted regularly to ensure consistency of the feature-effects with the variability model constraints during development. In this case, the runtime may be more important, as the analysis is executed many times. Additionally, the formula size may not be as relevant, as the consistency checking could be automated. Automated systems can deal better with larger formulas than humans. In this scenario, a lower simplification setting is advisable.

An open question remains why the runtime of the first product variant in the evaluation is so much higher than the second product variant. Both have a similar number of variability variables and feature-effect formula size. One possible explanation is that the formulas in the first product variant have a structure that is harder to simplify. The formulas may

have a structure where simplification opportunities cannot easily be found. If a condition that is hard to simplify appears in a parent conditional compilation block or file presence condition, the presence conditions of all the nested blocks also become hard to simplify. This could explain that the same number and size of feature-effect formulas for one product variant takes longer to simplify than for another one. We regrettably do not have access to the product variants to analyze this due to confidentiality reasons.

An interesting observation is that most of the runtime of the *AAS* setting is spent in a single step of the “equal sub-formula detection” approach. For the first product variant, more than 96 % of the time spent in the “combined approach” (**Measurement 3** for *AAS* setting) stems from step 4 (checking if formula changed) in the “equal sub-formula detection” approach (**Measurement 17**). For the second product variant, this ratio is 89 %. This is remarkable, as step 4 of the “equal sub-formula detection” approach is only a small utility step that checks if the preceding simplification changed the formula (i.e. if there was a simplification at all). It is not obvious why this step should take such a large portion of the overall runtime. This may indicate that the implementation has a bug in this step. Further investigation is required here, as fixing such a bug has the potential to greatly decrease the runtime of the *AAS* simplification setting.

## 5.4 srcML-Extractor

This section evaluates the srcML-extractor implementation for KernelHaven [KE19b], which is introduced in Chapter 4. The srcML-extractor parses C-source code files and creates a code model for them. The code model combines the syntactical information of a traditional abstract syntax tree (AST) with variability information (i.e. conditional compilation blocks implemented via the C-preprocessor). The srcML-extractor uses the srcML tool [CDM11; CMD+17] for the basic syntactical markup of the source code and transforms this markup to a code model for KernelHaven.

### 5.4.1 Setup

The srcML-extractor is evaluated on the Linux Kernel, a large open-source software product line [Lin19]. The Linux Kernel is the use-case of the MetricHaven infrastructure [EKS19], which the srcML-extractor was developed for. The latest Linux Kernel release as of writing, version 5.3.5, will be used (the same version is also used in the evaluation of the CodeBlockExtractor in Section 5.2). The srcML-extractor parses all .c files of the Linux Kernel individually and creates a code model for each of the files. The total number of these files will be measured (**Measurement 1**). Further measurements will be conducted to assess the *performance* and *robustness* of the srcML-extractor.

For the evaluation of the *performance*, the total runtime of the extraction process will be measured (**Measurement 2**). This includes the parsing of all source code files in the Linux Kernel. The srcML-extractor will be configured to use only a single thread, that means that files will be parsed after one another. Additionally, the runtime of the three steps performed in the srcML-extractor transformation process (cf. Figure 4.4) will be measured individually: XML parsing (**Measurement 3**), preprocessing (**Measurement 4**), and conversion (**Measurement 5**). As the XML parsing requires the complete output of the XML markup created by the srcML tool, this measurement includes the runtime of the srcML tool. The performance measurements will be performed on a virtual machine

running Ubuntu 16.04 with 40 logical CPU cores of an Intel Xeon E5-2650V3 @ 2.3 GHz and 314 GiB RAM. The measurement setup will be executed once as warmup. After that, three measurement executions will be performed and the one with the median execution time will be reported.

For the evaluation of the *robustness*, multiple measurements will be conducted to assess how many constructs the srcML-extractor cannot parse. These measurements rely on self-reported errors that the srcML-extractor detects during parsing. First, the number of unparseable files is measured (**Measurement 6**). These are cases where the srcML-extractor cannot parse the file at all. This may be due to constructs in the C-preprocessor that cannot be handled in the preprocessing step (cf. Section 4.3.1). Additionally, the number of files that contain an **ErrorElement** are measured (**Measurement 7**). If the srcML-extractor encounters a problem in the AST conversion step (cf. Section 4.3.2), it inserts a special **ErrorElement** that indicates that the transformation for this part of the AST has failed. The rest of the file, including the nested elements, are still transformed normally. The C-preprocessor condition parser component is re-used from the CodeBlock-Extractor (cf. Section 3.3). It is configured to handle Linux-specific macros, *fuzzy parsing* is enabled, and exceptions are thrown for all unparseable conditions. This means that unparseable C-preprocessor conditions will create **ErrorElements**.

The metrics in the MetricHaven framework are implemented on a per-function basis [EKS19]. As this is the use-case of the srcML-extractor, its *robustness* will also be measured on a per-function basis. For this, the total number of functions in the code model created by the srcML-extractor will be measured (**Measurement 8**). Another measurement will count how many of these functions contain an **ErrorElement** (**Measurement 9**). These measurements indicate how many functions can be supplied to the metric framework and how many of those contain inaccurate parsing results.

### 5.4.2 Results

The measurement results for the Linux Kernel version 5.3.5 are as follows: The total number of .c files in the Kernel is 27,416 (**Measurement 1**). Table 5.5 summarizes the results of the performance measurements. Timings of more than 1 minute are given in the format *minutes:seconds.milliseconds*. For the measurements that consist of multiple timings per execution, the minimum, median, average, maximum, and sum of the individual timings is given. **Measurement 2** is only a single timing, so this value is given in the sum column. The last column shows the relation of the sum column to the total runtime (**Measurement 2**).

	Min	Med	Avg	Max	Sum	Relative
<b>Measurement 2</b> (total extraction time)	—	—	—	—	18:55.861	100 %
<b>Measurement 3</b> (XML parsing)	0.003 s	0.023 s	0.038 s	1.817 s	17:41.676	95.5 %
<b>Measurement 4</b> (Preprocessing)	0.000 s	0.000 s	0.001 s	0.073 s	21.880 s	1.9 %
<b>Measurement 5</b> (Conversion)	0.000 s	0.000 s	0.001 s	0.209 s	40.644 s	3.6 %

Table 5.5: Runtime of srcML-extractor transformation steps

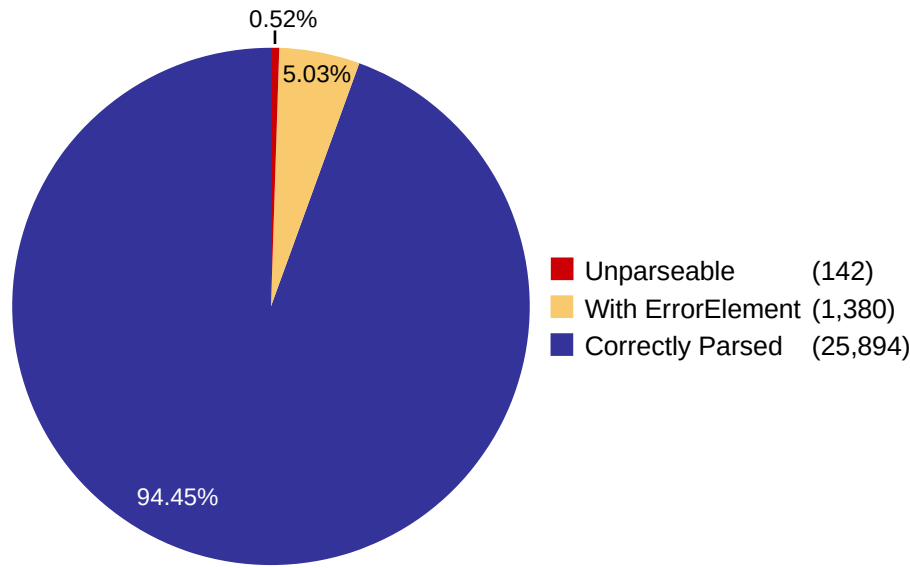


Figure 5.6: Ratio of files parseable by the srcML-extractor

Figure 5.6 shows the results of **Measurement 6** and **Measurement 7** in relation with **Measurement 1**. 142 files cannot be parsed at all (**Measurement 6**), that is about 0.52 % of the total number of files (**Measurement 1**). 1380 files contain an **ErrorElement** (**Measurement 7**), that is about 5.03 % of the total number of files (**Measurement 1**). The number of functions is also measured: 488,707 functions are in the files that were parsed (**Measurement 8**). Of these, 2,321 functions contain an **ErrorElement** (**Measurement 9**), which is about 0.47 % of the total amount of functions.

### 5.4.3 Discussion

The srcML-extractor parses C-source code files to a code model that combines a traditional abstract syntax tree (AST) representation with variability information from the C-preprocessor. This enables analyses that require a combined model of the syntactical structure of the code and the variability information. The specific use-case that motivated the development of the srcML-extractor is the MetricHaven framework. MetricHaven allows the implementation of variability-aware metrics for software product lines. For this use-case it is important that the code model created by the extractor closely represents the contents of the original source code file. This is because the metrics aim at quantifying aspects of the variability and the code that the developer has to work with.

The AST code model created by the srcML-extractor enabled the implementation of a large number of metrics in the MetricHaven framework [EK19]. As of the time of writing, 9 classes of code metrics have been implemented. Many of them have variations and allow to be combined with one or multiple of 7 variability weights. Variability weights are metrics for the variability model; their results can be used as weights in the calculation of the code metrics. Overall, this leads to a total of 42,796 individual metrics. This includes some traditional code metrics and a large number of variability-aware metrics designed for software product lines.

Most of the metrics in MetricHaven could be implemented straightforward with the AST created by the srcML-extractor. However, for some metrics the AST structure did not provide enough information. This is because the AST only represents the syntax down to the statement-level and leaves expressions as unparsed code strings. This can lead to problems for metrics that need information from the expressions-level. For instance, metrics that analyze the call graph require information when a function is called in an expression. With the AST structure of the srcML-extractor, this can only be implemented heuristically: if a name of a known function appears directly in front of an opening bracket then it is assumed that the function is called here. In practice, this approximation leads to good results.

The srcML-extractor has a back-off strategy for some constructs that it cannot parse. In such a case, an **ErrorElement** is created in the AST at the problematic location. **Measurement 6** and **Measurement 7** show that in total 1,522 files could not be parsed correctly. However, 1,380 (90.7 %) of these (**Measurement 7**) could still be handled due to the back-off strategy. Only 142 files (0.52 % of the total number of .c files) could not be parsed due to constructs that the preprocessing step cannot handle. The back-off strategy greatly decreases the amount of unparseable files, but it leads to files that contain incorrectly parsed parts (the **ErrorElements**). A flag in all parent elements of an **ErrorElement** allows the analysis to decide if it wants to consider these incorrectly parsed files.

The MetricHaven framework decides on a per-function basis whether to consider incorrectly parsed ASTs. The metrics are implemented on a per-function basis and the MetricHaven framework splits the ASTs up into individual functions. The user can configure whether functions that have the **ErrorElement** flag set should be considered or not. Even if the incorrectly parsed functions are discarded, the back-off strategy still has an advantage: if one function from a file cannot be correctly parsed, only that function is marked with the **ErrorElement** flag. All the other functions in the same file that are parsed correctly can still be considered in the metric computation. Without the back-off strategy, the whole file would need to be discarded and all functions, including the ones that can be parsed correctly, would be lost. **Measurement 8** shows that 488,707 functions are parsed by the srcML-extractor. Only 2,321 of these (about 0.47 %) contain an **ErrorElement** which a user may want to discard before the metric computation (**Measurement 9**).

The unparseable files and **ErrorElement** back-off strategy rely on the fact that the parser detects constructs that it cannot correctly transform. These self-reported errors do not fully reflect the *correctness* of the parser. There may be constructs that the srcML-extractor transforms without reporting an error, but the resulting code model is wrong. The example in Figure 4.12 in Section 4.3.1 shows a case where the srcML tool creates wrong markup. In this example, the srcML-extractor can transform the XML markup without an error, but the resulting model does not represent the source code correctly. During the development of the srcML-extractor, a small test-suite with common code constructs was created to check that these constructs are converted correctly. Additionally, manual inspections of the parsing result for real-world source code from the Linux Kernel were conducted throughout the development to check the correctness of the created models. The findings of these inspections were considered in the development of the srcML-extractor and are the basis for the illustrated edge-cases in Section 4.3.

**Measurement 2** through **Measurement 5** show the performance of the srcML-extractor implementation. The total runtime of about 18 minutes and 56 seconds (**Measurement 2**) means that, on average, parsing of each of the 27,416 .c files (**Measurement 1**) took

about 41.4 milliseconds. For comparison, this is two orders of magnitude slower than the CodeBlockExtractor, which takes 0.215 milliseconds per file (cf. Section 5.2). However, the CodeBlockExtractor only parses the C-preprocessor conditional compilation blocks and does not consider the C-code.

The XML parsing step of the srcML-extractor transformation process makes up about 95.5 % of the total runtime (**Measurement 3**). The XML parsing requires the complete output of the srcML tool, thus this measurement includes the runtime of the srcML tool. Also, as XML parsing usually has a high performance, most of the measured time for that step likely stems from waiting for the output of the srcML tool. The following preprocessing and conversion steps, the main parts of the srcML-extractor implementation, only make up a small amount of the total runtime (5.5 %, **Measurement 4** and **Measurement 5**). Thus the overhead of the transformation from srcML to a code model compatible with KernelHaven is relatively small.

## 5.5 Threats to Validity

This section discusses the threats to the validity of the evaluations in the previous sections. This discussion is combined for all previous sections as many points apply to more than one evaluation. It is split into three validity categories: internal, external, and construct validity. The fourth validity category, conclusion validity, has been left out. This is because conclusion validity is concerned with application of statistical analysis, but no statistical tests have been performed on the measured results. Additionally, for applied research like the evaluations conducted in this chapter, conclusion validity is considered to have the least priority [WRH+00, p. 74ff.]. The internal validity section discusses some points that could also be seen as related to conclusion validity.

### 5.5.1 Internal Validity

The internal validity of an evaluation represents that the measurement results are caused by the aspect that is intended to be measured [WRH+00, p. 63ff.]. This can be threatened by unintended influences while conducting measurements. For instance, if an evaluation compares two tools, the two tools have to be run under the same conditions. If other (external) influences are only present during the execution of one tool, the comparison of the two tools is inadequate.

Performance measurements are especially sensitive to unintended external influences, as many aspects may have an influence on modern computers. Other processes, the operating system (interrupts, etc.), and caches can, among other aspects, influence the momentary performance of the system. To mitigate some of these problems, most performance measurements were performed multiple times and the median is reported. While this does not *prevent* any external influences, it removes influences that cause individual measurements to be higher or lower than the “average” execution scenario. For the performance measurements of the simplification approaches in Section 5.3 it was regrettably not possible to execute the measurements multiple times. This is because the overall runtime of the evaluations was relatively long and the measurements were conducted by our industry partners at Bosch.

Most of the measurements in the evaluations were implemented by inserting probes into the evaluated component. For example, performance probes measure the elapsed time of a certain task. The measurement results of the individual probes are stored in-memory during the execution and are printed to the log file after the KernelHaven pipeline finished. The performance probes introduce side-effects into the performance measurements, as they use CPU and memory resources. They are implemented to be lightweight, but they nevertheless may influence the performance measurements, as a single KernelHaven pipeline execution can result in several tens of millions of individual probe measurements. While this factor is present during all executions of a specific evaluation, some of the evaluated settings contain more probes than others. For example, the *AAS* setting of the evaluation in Section 5.3 has more probes than the *Simple* setting.

The performance probes rely on the accuracy of the underlying time reporting that they use to calculate the elapsed time. In the evaluations here, the `System.nanoTime()` method of the Java virtual machine (JVM) is used to get timestamps for the start and end of a task. The elapsed time of the task is calculated by subtracting the start from the end. The documentation for the `nanotime` method only guarantees a precision of at least 1 millisecond [JAS14], although the JVM implementations used in the evaluations have higher precision. The aggregation of the individual probe measurements is implemented on the nanosecond scale that is returned by the `nanotime` method and the results are reported with millisecond precision.

In the evaluation of the non-Boolean transformation (Section 5.1) a considerable portion of the runtime is spent on copying files. The performance of file operations generally has a high variance. This is because storage hardware is typically orders of magnitudes slower than the CPU and RAM, and its performance largely depends on caches, which could not be controlled by us. Thus, the measured runtime of the preprocessing component, which copies a large amount of files, will be heavily influenced by these aspects. This is somewhat mitigated by the approach to execute the component four times and report the median runtime. However, there was no further analysis of the impact of file operations on the component as a whole.

The measurements conducted by our industry partners at Bosch (the simplification approach evaluation in Section 5.3 and the evaluation of the non-Boolean transformation on the PS-EC product line in Section 5.1) could not be controlled by us. We only prepared a KernelHaven pipeline with performance probes, which was then executed by our partners at Bosch. We could not control the circumstances of the execution, like the hardware or potential external influences. This could have had an effect on performance measurements. We instructed our partners at Bosch that performance measurements were conducted to partially prevent some negative influences. Another aspect here, that could not be controlled by us, is the selection of the product variants used for the evaluation. We instructed our partners at Bosch to use “representative” product variants. However, as the results in Section 5.3 show, there is a huge variance in the two selected product variants.

### 5.5.2 External Validity

The external validity of an evaluation represents how well the results are applicable outside of the specific evaluation [WRH+00, p. 63ff.]. This can be threatened by aspects in the evaluation that are not generalizable but only hold within the evaluated context. For example, if a tool is only evaluated on a certain type of input, the results may not apply to other types of input.

The evaluations for the non-Boolean transformation (Section 5.1) and the formula simplification approaches (Section 5.3) were only conducted on two product variants. At least for the formula simplification runtimes, these two product variants also show a relatively large variance. This small sample of product variants threatens the generalization of the evaluation results to other product variants. We instructed our industry partners at Bosch to select “representative” product variants, which slightly remedies the concerns here. However, a better measure to lessen this threat would be to evaluate on more product variants. Due to the high effort required for that, this was not possible in this thesis.

The formula simplification approaches (Section 5.3) were only evaluated in the context of the feature-effect analysis of the PS-EC product line. However, most of the simplification approaches introduced in Section 3.5 are general approaches in the sense that they can be applied to any arbitrary Boolean formula. The feature-effect formulas represent a special group here, as they are all created using the same formula (Formula 3.2). Thus, the feature-effect formulas that are used in the evaluation all have a similar structure. Additionally, the variability conditions of the product line, which the feature-effects are based on, are results of the non-Boolean transformation preprocessing (cf. Section 3.2). This is another source where similar structures are introduced systematically in the formulas. Because of these two points, the results of the evaluation cannot be easily generalized to arbitrary Boolean formulas.

The non-Boolean transformation was evaluated on generated test-cases, in addition to the two PS-EC product variants (Section 5.1). While this somewhat extends the external validity beyond the scope of the PS-EC product variant, the artificial generation is not based on any real-world product line. The goal of the test-case generation is to cover all transformation rules; it does not represent any distribution of variability in a real-world product line. Additionally, it only generates integer variability variables with a fixed range; it does not generate unrestricted variables as one would expect in a real-world product line. The evaluation with the generated test-cases gives some insight on the behavior of the underlying approach, but it does not generalize to other product lines with integer-based variability.

The CodeBlockExtractor was only evaluated on the Linux Kernel (Section 5.2). How many conditions the extractor can correctly parse largely depends on the kind of variability that is present in the product line. Thus, the specific percentages of this aspect of the evaluation do not generalize beyond the Linux Kernel. The further analysis of the unparseable conditions revealed that it is mostly caused by numeric operations and custom macro calls. This result can be transferred to other product lines: the CodeBlockExtractor can correctly parse pure Boolean variability. Additionally, the results of the runtime evaluation are expected to generalize to other product lines. This is because the runtime depends on the size of the product line and not on the kind of variability. The CodeBlockExtractor linearly runs through all source code files and parses each variability condition independently. This is in line with our qualitative observations when running the CodeBlockExtractor on the PS-EC product line. The ratio of unparseable conditions is different, but the runtime behavior is similar to the Linux Kernel.

The srcML-extractor was also only evaluated on the Linux Kernel (Section 5.4). The threats to external validity regarding the variability are the same as for the CodeBlockExtractor. Additionally, the srcML-extractor parses the C-source code. Here, the limitation to a single product line as the evaluation basis can also threaten the external validity. This is because the specific domain and coding conventions of the Linux Kernel can have

an influence on the constructs that appear in the C-code. Other projects may have conventions that cause certain constructs in the code to appear more or less often than in the Linux Kernel. This can have an effect on the *robustness* that was evaluated for the srcML-extractor.

### 5.5.3 Construct Validity

Construct validity of an evaluation represents how well the conducted measurements reflect the attribute that is intended to be evaluated [WRH+00, p. 63ff.]. This can be threatened by measurements that only partially (or not at all) reflect the intended abstract attribute. For instance, an evaluation intends to quantify the usability of a software product, but uses the number of buttons as a measurement. This measurement does not reflect the intended abstract attribute (usability).

A general threat to the the construct validity is that only the implementation can be measured, while the general approach is meant to be evaluated. This could lead to discrepancies when the measurements are influenced by implementation details that are not inherent to the theoretical approach itself. For instance, in the evaluation of the non-Boolean transformation (Section 5.1), a large amount of the total runtime is spent on copying files and waiting for the variability model. These tasks are implementation-specific decisions that are not inherent to the transformation approach presented in Section 3.2. To lessen this threat, we tried to take care to identify influences that stem from the implementation and not from the approach itself. The reported measurements are specific to the implementations, while the discussions argue which conclusions can be drawn for the underlying approaches.

Throughout all evaluations, the effort or overhead of the implementations is quantified by measuring the runtime. Other aspects that also could be considered to be “effort”, like the memory footprint, are not considered. This approach, that only runtime is measured to evaluate research prototypes, is common in our experience. Additionally, no other aspect that constitutes “effort” was conspicuous in typical applications of the implementations. If, for example, we would have encountered memory problems, a memory measurement would have been included in the evaluation.

In the evaluation of the non-Boolean transformation (Section 5.1), the size of the condition formulas is measured as the number of characters in the C-preprocessor line. The length of the formula string is only a rough estimation of the size. It is mainly influenced by the number of variables, as these are typically many characters long. In contrast, the number of operators only has little effect on the length of the string. Additionally, variables that happen to have a longer name influence the formula string length more than variables with shorter names. A better measurement here would be the number of operators or number of variables, like the measurements for the formula size in Section 5.3. However, the formula string length was chosen as it is easy to implement with relatively little overhead. Additionally, the ratio of the pre- and post-transformation condition lengths is reported to indicate the growth rate, which makes the absolute length of the conditions less relevant.

The evaluations of the non-Boolean transformation (Section 5.1), the CodeBlockExtractor (Section 5.2), and the srcML-extractor (Section 5.4) measure the number of elements that could not be transformed or parsed. This is used to indicate how *robust* the implementations are against the variety of possible inputs. This is measured by counting how

many exceptions the implementations themselves report. This measurement does not include cases where the implementations *should* raise an exception, but instead incorrectly transform or parse the element. The number of conditions where the implementations fail to correctly transform or parse the element may be higher than the measured results. Thus the reported robustness may be too high. To lessen this threat, we implemented the approaches defensively to throw exceptions in all cases where it is not clear that the input can be handled correctly. Additionally, the test-suites that are developed together with the implementations aim at a high coverage of different input classes to ensure that all successfully transformed or parsed elements are correct.

The formula simplification approaches, evaluated in Section 5.3, aim at increasing the *understandability* of the feature-effect formulas. As discussed in Section 3.5, this problem is reduced to the size of the propositional formulas. This is because the formulas are constructed using only three simple operators ( $\wedge$ ,  $\vee$ , and  $\neg$ ), parenthesis, variables, and constants (*true* and *false*). However, *understandability* is a quality where human perception and cognitive abilities play a major role. There may be aspects involved beyond simply the size of the formula. A study with human subjects that evaluates the understandability of the simplified and unsimplified formulas will result in a more accurate representation. However, this greatly increases the effort required for this evaluation and is not feasible in the context of this thesis. We believe that the approximation using the formula size is reasonable in the evaluation of the simplification approaches. This is also in line with the requirements of our industry partners at Bosch, who said that the original size of the feature-effect formulas was the main obstacle for their domain experts.

Another threat to the construct validity of the evaluation of the simplification approaches is that the approaches were not evaluated individually. Instead, the settings offered by the KernelHaven infrastructure are used, which combine different approaches. For example, the “simplifying disjunction” approach is enabled in three of the settings: *Simplifying Disjunction*, *Visitor*, and *AAS*. This closely represents real-world use-cases, where multiple of the approaches are combined to simplify the resulting feature-effect formulas. However, it limits drawing conclusions for the individual simplification approaches.

## 6 Conclusion

This thesis presented two approaches for statically analyzing C-source code with variability. The common focus was on the extraction and representation of the necessary information from the source code artifacts. The two approaches have different goals with respect to which kind of information is extracted. They enable different kinds of analyses based on their created code models.

The first approach is the analysis of C-preprocessor variability in a software product line (cf. Section 3). The goal of this analysis is to extract domain knowledge that is implicitly encoded in the variability in the source code. It is based on the *feature-effect* approach by Nadi et al. [NBK+14]. The result of this analysis are preconditions for the variability variables that define when they have an *effect* on the product generation. This means, that the preconditions encode when changing the configuration of a variability variable also changes the product that is generated.

The use-case for this approach was the industrial Bosch PS-EC product line (cf. Section 3.1). The goal for this use-case was that the preconditions created by the feature-effect analysis serve as a basis for the creation of variability model constraints for this product line. The application of the analysis identified independent and dependent variability variables (cf. Section 3.6). The independent variables always have an effect on the product derivation, regardless of the configuration of the other variability variable. The dependent variability variables have a precondition that defines under which circumstances they influence the final product.

The analysis pipeline implementation for this approach consists of several steps. First, the integer-based variability used in the PS-EC product line is converted to propositional logic (cf. Section 3.2). This is required as the *feature-effect* approach and the existing tooling only work with Boolean formulas. Then, a parser creates a model of the C-preprocessor variability in the source code files (cf. Section 3.3). This model is a hierarchy of the conditional compilation blocks with their presence conditions. Combined with other models of variability from different assets of the product line, it is then analyzed with the feature-effect approach. This approach had to be slightly adapted to this use-case, as the transformation of the integer-based variability introduces some constraints on the propositional replacements (cf. Section 3.4). Finally, the results computed by the feature-effect approach need to be simplified.

The simplification of the resulting formulas is required as they are meant to be understandable by the domain experts (cf. Section 3.5). Established approaches and tools for Boolean formula simplification are not applicable as their runtime behavior and assumptions on the formula structure are not feasible. Thus, several heuristic approaches have been developed that reduce the size of the resulting formulas. Some of them are generic, that is they can simplify arbitrary Boolean formulas, while others are specific to the feature-effect computation. The evaluation showed that the simplification step required the most runtime in the analysis pipeline (cf. Section 5.3). Generally, the more time is invested in the simplification the smaller the formulas become.

This analysis pipeline demonstrates that there is useful domain knowledge encoded in the variability implementation in a real-world product line. Extraction of this promises useful applications, like creation of variability model constraints, improvement of the configuration process, and detection of mismatches between variability model constraints and implementation. Future work could look at further applications of the computed feature-effects, for example to check adherence to architectural constraints. Another possible direction can be exploring alternative analysis concepts to extract domain knowledge from the variability in the implementation artifacts. One possible idea could be to analyze how much a variability variable changes the final product. This could be used to classify the variability variables into ones that have a large impact (e.g. variables that control whole features) and others that only modify small aspects (e.g. variables that modify a small aspect within a larger feature).

Another finding from the application of this approach is that analyses have to be adapted to the specific product line. Variability is implemented differently in different product lines, so at least the variability extractors have to be adapted to this. It may also be required to adapt the analysis itself. In the case of the Bosch PS-EC product line, the integer-based variability was converted to propositional logic. In addition to changes in the extraction process, this also introduced a new aggregation step to the feature-effect computation (cf. Section 3.4). Another aspect where analyses have to be adapted to product lines is the types of assets that need to be analyzed. In the Bosch PS-EC use-case we found that considering more sources of variability information improves the results. Thus, analyses have to consider which artifact types exist in a product line and how their variability information can be integrated into the analysis process. Future work in this area can adapt existing analysis approaches to more real-world use-cases. This broadens the knowledge of possible adaptations and may lead to a generic model of variability information sources that analyses can use.

The second approach presented in this thesis is a parser for C-source code that combines the syntactical structure of the code with variability information (cf. Chapter 4). The result is an abstract syntax tree (AST) representation of the code with additional nodes that represent the C-preprocessor variability. This enables static analyses that require the information from both, the code structure and the variability. The specific use-case for the implementation in this thesis is a framework for computing variability-aware metrics for software product lines (cf. Section 4.1). This use-case requires that the code model closely reflects the content of the original source code file.

The parser is based on the srcML tool [CDM11; CMD+17], which creates low-level syntactical markup while leaving the original content of the source file intact (cf. Section 4.2). The approach developed in this thesis integrates the variability information into the hierarchical AST structure of the code and transforms it into a combined code model (cf. Section 4.3). In this process, a challenge arises from *undisciplined* C-preprocessor directives, i.e. directives that go against the structure of the code AST. Representing this with the constraint that the resulting model closely reflects the source code structure requires special transformation rules (cf. Section 4.3.1). However, some constructs still cannot be transformed by this approach.

The AST that is created by the parser only represents the syntactical structure down to the statement-level (cf. Section 4.3.2). More fine-grained information, like expressions, are left as unparsed code strings. This reduced the required development effort to create the data structure and transformation process for the AST. Additionally, it makes the parser more accepting as syntactical errors in the expressions can be ignored. However, this has

the disadvantage that analyses that require expression-level syntactical information cannot easily be implemented with this code model. In the use-case of the metric framework, a few metrics had to be implemented heuristically on the code strings of expressions.

Three alternative approaches that also build combined models of source code with variability information were explored (cf. Section 4.4). These approaches do not fulfill the requirement of the metric framework that the code model should closely reflect the source file content. Two of the approaches change the AST structure of the code so that the integration of variability is easier. This results in syntactically correct ASTs, but it diverges from the original source code content. The third approach does not integrate undisciplined C-preprocessor directives into the AST structure at all.

The parser developed in Chapter 4 does not implement any analysis itself. Instead, it enables new static analysis approaches to be developed based on the created AST. The AST structure offers a model that combines the syntactical structure of the code with the variability information that changes code depending on the product configuration. This representation is very close to the contents of the original source code file. This allows analyses to consider the developer perspective on the source code containing variability. Analyses developed with this AST will likely aim at supporting the developer working with the source, for example with regard to understandability. For instance, there is ongoing work with the variability-aware metric framework with the goal to detect how variability influences the amount of errors in the code.



## Bibliography

- [ABK+13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-oriented software product lines: concepts and implementation*. Springer Science & Business Media, 2013. DOI: 10.1007/978-3-642-37521-7.
- [BHH+82] Robert K. Brayton, Gary Hachtel, L. H. Hemachandra, Richard Newton, and Alberto Sangiovanni-Vincentelli. “A comparison of logic minimization strategies using ESPRESSO: An APL program package for partitioned logic minimization”. In: *Proceedings of the International Symposium on Circuits and Systems*. 1982, pp. 42–48.
- [Bry86] Randal E. Bryant. “Graph-based algorithms for boolean function manipulation”. In: *IEEE Transactions on Computers* C-35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819.
- [BSL+13] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. “A study of variability models and languages in the systems software domain”. In: *IEEE Transactions on Software Engineering* 39.12 (2013), pp. 1611–1640. DOI: 10.1109/TSE.2013.34.
- [CDM11] Michael L. Collard, Michael J. Decker, and Jonathan I. Maletic. “Lightweight transformation and fact extraction with the srcML toolkit”. In: *2011 IEEE 11th international working conference on source code analysis and manipulation*. IEEE. 2011, pp. 173–184. DOI: 10.1109/SCAM.2011.19.
- [CMD+17] Michael L. Collard, Jonathan I. Maletic, Michael J. Decker, et al. *srcML Tool*. <https://www.srcml.org/>. Last visited: 21.10.2019. 2017.
- [Col05] Michael L. Collard. “Addressing source code using srcml”. In: *IEEE International Workshop on Program Comprehension Working Session: Textual Views of Source Code to Support Comprehension (IWPC’05)*. Citeseer. 2005.
- [EDK+18] Sascha El-Sharkawy, Saura Jyoti Dhar, Adam Krafczyk, Slawomir Duszynski, Tobias Beichter, and Klaus Schmid. “Reverse Engineering Variability in an Industrial Product Line: Observations and Lessons Learned”. In: *Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC’18)*. Vol. 1. - ACM, 2018, pp. 215–225. DOI: 10.1145/3233027.3233047.
- [EK19] Sascha El-Sharkawy and Adam Krafczyk. *MetricHaven plug-in*. <https://github.com/KernelHaven/MetricHaven>. Last visited: 21.10.2019. University of Hildesheim, Software Systems Engineering, 2019.
- [EKS17] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. “An empirical study of configuration mismatches in linux”. In: *Proceedings of the 21st International Systems and Software Product Line Conference- Volume A*. ACM. 2017, pp. 19–28. DOI: 10.1145/3106195.3106208.

- [EKS19] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. “MetricHaven - More Than 23,000 Metrics for Measuring Quality Attributes of Software Product Lines”. In: *Proceedings of the 23rd International Systems and Software Product Line Conference*. Vol. B. ACM, 2019. DOI: 10.1145/3307630.3342384.
- [EYS18] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. “Metrics for analyzing variability and its implementation in software product lines: A systematic literature review”. In: *Information and Software Technology* (2018). DOI: 10.1016/j.infsof.2018.08.015.
- [GBA+19] Sten Grüner, Andreas Burger, Hadil Abukwaik, Sascha El-Sharkawy, Klaus Schmid, Tewfik Ziadi, Anton Paule, and Felix Suda. “Demonstration of Tool Chain for Feature Extraction, Analysis and Visualization on an Industrial Case Study”. In: *Proceedings of the 17th IEEE International Conference on Industrial Informatics*. 2019.
- [GS19] Lea Gerling and Klaus Schmid. “Variability-aware semantic slicing using code property graphs”. In: *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*. ACM. 2019, p. 12. DOI: 10.1145/3336294.3336312.
- [JAS14] *Java API System.nanoTime Documentation*. <https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime>. Last visited: 21.10.2019.
- [KE+19] Adam Krafczyk, Sascha El-Sharkawy, et al. *KernelHaven infrastructure repository*. <https://github.com/KernelHaven/KernelHaven>. Last visited: 21.10.2019. University of Hildesheim, Software Systems Engineering, 2019.
- [KE17] Adam Krafczyk and Sascha El-Sharkawy. *UndertakerExtractor plug-in*. <https://github.com/KernelHaven/UndertakerExtractor>. Last visited: 21.10.2019. University of Hildesheim, Software Systems Engineering, 2017.
- [KE18a] Adam Krafczyk and Sascha El-Sharkawy. *KbuildMinerExtractor plug-in*. <https://github.com/KernelHaven/KbuildMinerExtractor>. Last visited: 21.10.2019. University of Hildesheim, Software Systems Engineering, 2018.
- [KE18b] Adam Krafczyk and Sascha El-Sharkawy. *KconfigReaderExtractor plug-in*. <https://github.com/KernelHaven/KconfigReaderExtractor>. Last visited: 21.10.2019. University of Hildesheim, Software Systems Engineering, 2018.
- [KE18c] Adam Krafczyk and Sascha El-Sharkawy. *NonBooleanUtils plug-in*. <https://github.com/KernelHaven/NonBooleanUtils>. Last visited: 21.10.2019. University of Hildesheim, Software Systems Engineering, 2018.
- [KE19a] Adam Krafczyk and Sascha El-Sharkawy. *FeatureEffectAnalysis plug-in*. <https://github.com/KernelHaven/FeatureEffectAnalysis>. Last visited: 21.10.2019. University of Hildesheim, Software Systems Engineering, 2019.
- [KE19b] Adam Krafczyk and Sascha El-Sharkawy. *srcML-Extractor plug-in*. <https://github.com/KernelHaven/srcMLExtractor>. Last visited: 21.10.2019. University of Hildesheim, Software Systems Engineering, 2019.

- [KES18a] Adam Krafczyk, Sascha El-Sharkawy, and Klaus Schmid. “Reverse engineering code dependencies: converting integer-based variability to propositional logic”. In: *Proceedings of the 22nd International Conference on Systems and Software Product Line-Volume 2*. ACM. 2018, pp. 34–41. DOI: 10.1145/3236405.3237202.
- [KES18b] Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. “KernelHaven: an experimentation workbench for analyzing software product lines”. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*. ACM. 2018, pp. 73–76. DOI: 10.1145/3183440.3183480.
- [KES18c] Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. “KernelHaven: an open infrastructure for product line analysis”. In: *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 2*. ACM. 2018, pp. 5–10. DOI: 10.1145/3236405.3236410.
- [KGR+11] 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: *ACM SIGPLAN Notices* 46.10 (2011), pp. 805–824. DOI: 10.1145/2076021.2048128.
- [KKH+10] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. “TypeChef: toward type checking #ifdef variability in C”. In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*. ACM. 2010, pp. 25–32. DOI: 10.1145/1868688.1868693.
- [Kra18] Adam Krafczyk. *CodeBlockExtractor plug-in*. <https://github.com/KernelHaven/CodeBlockExtractor>. Last visited: 21.10.2019. University of Hildesheim, Software Systems Engineering, 2018.
- [Lin19] *Linux Kernel Archive*. <https://www.kernel.org/>. Last visited: 21.10.2019.
- [LKA11] Jörg Liebig, Christian Kästner, and Sven Apel. “Analyzing the discipline of preprocessor annotations in 30 million lines of C code”. In: *Proceedings of the tenth international conference on Aspect-oriented software development*. ACM. 2011, pp. 191–202. DOI: 10.1145/1960275.1960299.
- [McC56] Edward J. McCluskey Jr. “Minimization of Boolean functions”. In: *Bell system technical Journal* 35.6 (1956), pp. 1417–1444. DOI: 10.1002/j.1538-7305.1956.tb03835.x.
- [McC76] Thomas J. McCabe. “A complexity measure”. In: *IEEE Transactions on software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.
- [MRG+17] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Balduino Fonseca. “Discipline matters: Refactoring of preprocessor directives in the #ifdef hell”. In: *IEEE Transactions on Software Engineering* 44.5 (2017), pp. 453–469. DOI: 10.1109/TSE.2017.2688333.
- [NBK+14] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. “Mining configuration constraints: Static analyses and empirical results”. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 140–151. DOI: 10.1145/2568225.2568283.

- [PS08] Hendrik Post and Carsten Sinz. “Configuration lifting: Verification meets software configuration”. In: *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society. 2008, pp. 347–350. DOI: 10.1109/ASE.2008.45.
- [Qui52] Willard V. Quine. “The problem of simplifying truth functions”. In: *The American mathematical monthly* 59.8 (1952), pp. 521–531. DOI: 10.1080/00029890.1952.11988183.
- [Qui55] Willard V. Quine. “A way to simplify truth functions”. In: *The American Mathematical Monthly* 62.9 (1955), pp. 627–631. DOI: 10.1080/00029890.1955.11988710.
- [SEK19] Klaus Schmid, Sascha El-Sharkawy, and Christian Kröher. “Improving Software Engineering Research Through Experimentation Workbenches”. In: *From Software Engineering to Formal Methods and Tools, and Back*. Springer, 2019, pp. 67–82. DOI: 10.1007/978-3-030-30985-5\_6.
- [SHF07] *SPLC Hall of Fame member Bosch*. <https://splc.net/fame/bosch/>. Last visited: 21.10.2019. SPLC, 2007.
- [STB+04] Mirjam Steger, Christian Tischer, Birgit Boss, Andreas Müller, Oliver Pertler, Wolfgang Stolz, and Stefan Ferber. “Introducing PLA at Bosch Gasoline Systems: Experiences and practices”. In: *International Conference on Software Product Lines*. Springer. 2004, pp. 34–50. DOI: 10.1007/978-3-540-28630-1\_3.
- [STL+10] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. “Efficient extraction and analysis of preprocessor-based variability”. In: *ACM SIGPLAN Notices*. Vol. 46. 2. ACM. 2010, pp. 33–42. DOI: 10.1145/1942788.1868300.
- [TAK+14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. “A classification and survey of analysis strategies for software product lines”. In: *ACM Computing Surveys (CSUR)* 47.1 (2014), p. 6. DOI: 10.1145/2580950.
- [TBM+12] Christian Tischer, Birgit Boss, Andreas Müller, Andreas Thums, Rajneesh Acharya, and Klaus Schmid. “Developing long-term stable product line architectures”. In: *Proceedings of the 16th International Software Product Line Conference-Volume 1*. ACM. 2012, pp. 86–95. DOI: 10.1145/2362536.2362551.
- [TLS+11] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. “Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem”. In: *Proceedings of the sixth conference on Computer systems*. ACM. 2011, pp. 47–60. DOI: 10.1145/1966445.1966451.
- [VSR07] Frank J. Van der Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Business Media, 2007. DOI: 10.1007/978-3-540-71437-8.
- [WRH+00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering. An Introduction*. Kluwer Academic Publishers, 2000. DOI: 10.1007/978-1-4615-4625-2.

- [YGA+14] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. “Modeling and discovering vulnerabilities with code property graphs”. In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 590–604. DOI: 10.1109/SP.2014.44.
- [ZWN+06] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A Vouk. “On the value of static analysis for fault detection in software”. In: *IEEE transactions on software engineering* 32.4 (2006), pp. 240–253. DOI: 10.1109/TSE.2006.38.