OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

INF FAKULTÄT FÜR INFORMATIK

Analysis Techniques to Support the Evolution of Variant-Rich Software Systems

Habilitationsschrift

zur Erlangung der Venia legendi für das Fach

Informatik

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

von: Dr.-Ing. Sandro Schulze

geb. am 04.06.1980 in Osterburg/Altmark

Gutachterinnen/Gutachter:

Prof. Dr. Gunter Saake
Prof. Dr. Sven Apel
Prof. Dr. Thorsten Berger

Magdeburg, den 26.06.2019

University of Magdeburg

School of Computer Science

Habilitation

Analysis Techniques to Support the Evolution of Variant-Rich Software Systems

Author:

Dr.-Ing. Sandro Schulze

Day of Submission: March 19, 2019

Schulze, Dr.-Ing. Sandro:
Analysis Techniques to Support the Evolution of Variant-Rich Software Systems
Habilitation, University of Magdeburg, 2019.

Abstract

Software is eating the world is a common saying nowadays, paraphrasing the fact that software influences almost all parts of industry or our daily life. This makes software subject to mass production while at the same time, there is an increasing demand for customization of software systems to adhere to specific requirements of users or environments. As a result, a software system is developed in many similar, yet different variants, thus, constituting a *variant-rich software system*. To enable development of such variant-rich systems, two approaches are commonly used: First, *structured reuse* by means of *integrated variability* allows to specify the differences between variants on domain and implementation level. Second, *adhoc reuse* by means of *clone-and-own*, which reuses existing variants by copying them and afterwards, apply the required modifications (e.g., code changes) to achieve the desired variant. While both are commonly used, they raise challenges to the evolution of variant-rich system for aspects such as maintainability, testability, extensibility or reliability.

In my research, I have developed techniques that allow to understand the reasons behind evolutionary challenges and how to identify and mitigate them. In this thesis, I summarize this research, mainly conducted within the last 4 years. In particular, I will present empirical as well as evolutionary analysis techniques to better understand the problems that integrated variability cause for developers and system quality. For clone-and-own systems, I will present reverse engineering techniques for different artefact types (models, requirements) that aim to extract the variability information, and thus, provide developers with a global and unified view of all variants under development.

Zusammenfassung

Software ist auf dem Vormarsch. Ob durch voranschreitende Technologien, eine zunehmende Automatisierung oder die vielbeschworene Digitalisierung; Software beeinflusst mittlerweile fast alle Bereiche der Industrie oder unseres täglichen Lebens. Dies macht Software zu einem Gegenstand der Massenproduktion, während gleichzeitig die Nachfrage nach Anpassungen von Softwaresystemen an spezifische Anforderungen von Benutzern oder Umgebungen zunimmt. Infolgedessen wird ein Softwaresystem in vielen ähnlichen, aber unterschiedlichen Varianten entwickelt und bildet somit ein *variantenreiches Softwaresystem*. Um die Entwicklung derartiger variantenreicher Systeme zu ermöglichen, werden im Allgemeinen zwei Ansätze verwendet: Erstens ermöglicht die *strukturierte Wiederverwendung* mittels integrierter Variabilität, die Unterschiede zwischen Varianten auf Domänen- und Implementierungsebene zu definieren. Zweitens, *Ad-hoc Wiederverwendung* mittels *clone-and-own*, wobei vorhandene Varianten durch Kopieren wiederverwendet werden und anschließend die erforderlichen Modifikationen (z. B. Codeänderungen) auf der kopierten Variante durchgeführt werden, um das gewünschte Ergebnis zu erreichen. Obwohl beide häufig verwendet werden, stellen sie hinsichtlich der Wartbarkeit, Testbarkeit, Erweiterbarkeit oder Zuverlässigkeit die Entwicklung variantenreicher Systeme vor große Herausforderungen.

In meiner Forschung habe ich Techniken entwickelt, die es ermöglichen, die Gründe für diese evolutionären Herausforderungen zu verstehen, zu erkennen und zu begrenzen. In dieser Arbeit fasse ich diese Forschung zusammen, die hauptsächlich in den letzten 4 Jahren durchgeführt wurde. Insbesondere werde ich sowohl empirische als auch evolutionäre Analysetechniken vorstellen, um die Probleme, die die integrierte Variabilität für Entwickler und die Systemqualität verursacht, besser zu verstehen. Für *clone-and-own* Systeme werde ich Reverse Engineering Techniken für verschiedene Artefakttypen (Modelle, Anforderungen) vorstellen, die darauf abzielen, die Informationen zur Variabilität zu extrahieren, und so Entwicklern eine globale und einheitliche Ansicht aller in der Entwicklung befindlichen Varianten bieten.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

In this chapter, I provide an overview of this thesis. In particular, I explicate the motivation for my research of the last five years, summarize the objectives that are behind this research, and present an outline of the remaining chapters.

1.1 Motivation

Software is eating the world is a a common saying, paraphrasing the observation that software influences almost all parts of industry or our daily life. To give an intuition what this means, just think of smartphones that are owned by almost each human on the planet; looking further, we see the rapid development of *smartX* technologies that make software an integral and imperative part of traffic (e.g., car, infrastructure), communication, but also of living (e.g., by smart home technology). And, last but not least, the still increasing success of embedded systems that constitute most of the software systems in the world and culminate in popular and fast-growing areas such as *cyber-phycial systems (CPS)*. In short, whole domains are shifting towards a software-based economy, which leads to a high demand of software systems.

This demand and pervasiveness of software makes it subject to mass production while at the same time, there is an increasing demand for customization of software systems to adhere to specific requirements of users or environments. To cope with this customization and provide tailor-made software systems that fit exactly the needs of respective stakeholders, the concept of *variant-rich software systems*[1] has been proposed. The core idea of this concept is that, instead of only one single software system, a whole family of software systems is developed with each variant being similar yet different from each other [50, 10, 3].

[1]Other terms commonly used are *Software Product Line (SPL)*, variable software system, and highly-configurable software system, respectively.

To develop such variant-rich systems, basically two approaches exist: *structured & planned reuse* and *adhoc reuse*.
For **structured & planned reuse**, the core concept is that multiple variants are developed simultaneously. To this end, *variability* is introduced by predefined *configuration options* that enable stakeholders to tailor the artifacts of a variant, such as code or models, according to the requirements. A common way to achieve this kind of reuse is SPLE, a development paradigm that enables the development of all artifacts and for all variants in a structured and preplanned way [9]. To express this variability on artifact level, different *integrated variability* mechanisms exist such as preprocessor directives, *Feature-Oriented Programming (FOP)* [53, 4], *Delta-Oriented Programming (DOP)* [57], *Aspect-Oriented Programming (AOP)* [28], but also plugins or frameworks. More details about how such mechanisms work is provided in Chapter 2. As a result of integrated variability, both, artifacts that are distinct between variants but also those who are common across variants are known to developers and other stakeholders, and thus, this information can be taken into account for evolving such kind of variant-rich software systems.

For **adhoc reuse**, no dedicated development paradigm is necessary or even exists. Instead, the similarity between an existing software system and a new one is employed by copying all relevant artifacts of the existing system and adapt them to meet the new requirements [11]. This process is commonly referred to as *clone-and-own* [14, 55] and while its realization may take place in different ways, it encompasses basically three steps. First, a software system that contains a considerable amount of reusable artifacts must be identified. The more artifacts can be reused for the new system, the better as this minimizes the effort for adapting the original system. Second, the artifacts (all or only the reusable parts) of the original system are copied to the new variant. At this point, the new variant is created and considered to have its own life, independent of the original software system. Third, the new variant is modified to address the corresponding requirements, in particular those that are not covered by the reused artifacts. As a result of this process, the variability is not made explicit, and thus, differences and commonalities between variants are not known to any stakeholder. Hence, there is a lack of information about *how* and *where clone-and-own* variants differ from each other.

Disregarding their differences, both approaches for developing variant-rich systems share one common aspect: The developed systems tend to be of certain complexity and long-living, that is, they are planned to live for years or even decades. Hence, these systems are subject to ongoing changes on all levels (e.g., architecture, artifacts), which is commonly referred to as *software evolution* [33]. While this evolution is necessary to keep the systems flexible, extensible and up-to-date with respect to other components or environments, it is also prone to negatively affect quality attributes of the evolving software system, known as *software aging* [51]. Among the attributes defined by the ISO/IEC 9126 standard [63], maintainability, changeability, testability, or extensibility but also reliability or functionality are likely to be affected by software evolution. Consequently, future changes are impeded, leading to a devil's circle resulting into a fragile and unmanageable software system. As an additional challenge to software evolution, variant-rich systems add another dimension to the problem, that is, the respective reuse approach that may cause additional challenges to evolve the underlying system while keeping the quality high. This, in turn, leads to the inevitable need to control the evolution of variant-rich system and to understand how *integrated variability* as well as *clone-and-own* contribute to the decay of this system during evolution.

1.2 Objectives

In this thesis, I summarize my research of the last five years by detailing selected papers and their contribution. All of this work contributes to a larger goal that consists of the following, interrelated two parts:

Understanding the influence of reuse mechanisms on software evolution.
To guide developers or other stakeholders through the evolutionary process of variant-rich software systems, it is imperative to understand how the reuse mechanism affects this process. To this end, insights are necessary that go beyond existing wisdom, but come with resilient data and other information that expose the existing problems. Parts of my research address this question based on the development history, pattern identification, or statistical reasoning, but also by means of empirical methods, in particular, to understand the *socio-technical aspects* (i.e., developers' perception of the considered reuse mechanisms) that results from variant-rich software systems. I argue that without understanding the impact of reuse mechanisms on software evolution, it is impossible to find appropriate countermeasures or at least create an awareness of critical parts of the software system with respect to software quality.

Developing techniques to analyze and counter the impact of reuse mechanisms on evolution. To gain a comprehensive understanding, as envisioned above, it is necessary to analyze variant-rich systems with respect to (A) the reuse mechnism applied and (B) the (quality) attributes affected by evolution. To this end, the second part of my research encompasses the development of analysis techniques for variant-rich software systems. These techniques are tailored to integrated variability and *clone-and-own*, respectively and include evolutionary analyses, statistical analyses, static analyses (e.g., software metrics), but also techniques to extract variability from different development artifacts such as source code or models. As a result, theses techniques leverage a vast amount of information that allows to reason about how the evolution of variant-rich software systems takes place, and thus, to understand the challenges that have to be addresses along with this evolution the ensure the respective quality.

1.3 Outline

The remainder of this thesis is structured as follows.

Chapter 2. In this chapter, I provide foundational information that is necessary to understand the following chapters. In particular, I introduce SPLE in greater details, explain selected variability mechanisms, and how *clone-and-own* is commonly applied. Finally, I will *highlight the challenges of evolving variant-rich software systems.*

Chapter 3. In this chapter, I present techniques to analyze the impact of integrated variability on software evolution and software quality. To this end, I summarize the following contributions: First, I introduce the notion of *variability-aware code smells* as a mean to define in appropriate usage of certain variability implementation mechanisms. Then, I introduce a technique and its evaluation to detect such smells in a flexible and scalable way. Finally, I present results of an empirical study that evaluates how sich smells may affect the maintainability of variant-rich software systems.

Chapter 4. I this chapter, I present techniques to analyze variant-rich systems developed with *clone-and-own*, mainly focussing on extraction of variability information. To this end, I will summarize contributions that comprise two different artifact types for the analysis. First, I introduce a model-based technique for variability mining in block-based models, mainly MATLAB/Simulink models and state charts. Moreover, I present a more general similarity analysis for MATLAB/Simulink that allows for an n-way comparison, and thus, is highly efficient even for large-scale models. Second, I present techniques that rely on deep learning and natural language processing to extract variability from natural language requirements.

Chapter 5. In this chapter, I summarize my contributions and their impact on the field of research. Moreover, I highlight open challenges and how to address them in future.

Appendix A. In the appendix, I list all papers I summarized in the chapters above and point out my personal contribution for each of these papers.

1.4 A Note from the Author

Writing a monography, and a thesis in particular, always raises the question which grammatical person should be used. Clearly, the thesis itself is a single-authored piece of work, with at most some reviewing and comments by others. Consequently, using “I” (i. e., *first person singular*) is the most suitable grammatical person to be used. However, the research presented in a thesis is usually not conducted in isolation; rather, it is the result of fruitful collaborations. Hence, this would justify using “we” (i. e., *first person plural*) as a grammatical person.

In this thesis, I finally solve this issues as follows: For chapters that are unique to this thesis such as the introduction (Chapter 1) or the background (Chapter 2), I use the first person singular. This may also apply to the introduction of other chapters, where I explain aspects that refer to this thesis, and thus, have no relation to any paper. In contrast, for chapters/content that refer to research conducted with others (i. e., Chapter 3, Chapter 4, and parts of Section 2.4), I use the first person plural.

Credit where credit is due.

Sandro Schulze

2. Background

This chapter introduces the basic notions, relevant in this thesis, and highlights the challenges that I have addressed with my research of the recent years. In particular, I introduce a methodology for structured reuse, introduce variability implementation mechanisms, relevant for my research, and provide information about adhoc reuse in practice. Finally, I explicate challenges that arise from structured and adhoc reuse, respectively, for evolving variant-rich systems.

2.1 Structured Reuse with Software Product Line Engineering

As mentioned in Chapter 1, software plays more and more a pivotal role in many domains, resulting into a software-based economy that is pervasive. As a result, the amount of software systems required is steadily increasing, thus, giving rise to *mass customization* of software [32]. This development also affects the way *how* software is developed today. While in earlier years software systems have been developed for a specific purpose (with a fixed set of requirements), they are now supposed to be used in changing environments (e.g., different hardware) or with different users, and thus, must adhere to changing requirements over time.

As an example, consider the LINUX kernel, which is the foundation for many variants of the LINUX operating system (called distribution).[1] There are distributions for a vast amount of domains such as Desktop PC, mobile devices, mainframe computers, embedded devices, and so forth. Moreover, the LINUX kernel supports a large variety of platforms by means of processor architectures. A simple search on distrowatch.com reveals that there are several hundreds of LINUX distributions under *active* development, which each of them using the same kernel, but in different, very tailor-made ways

[1]https://www.kernel.org/linux.html

(i.e., different set of functionalities) . Obviously, developing an own linux kernel for each of these distributions from scratch not only wastes resources such as developers or time; it also makes it infeasible to react on new requirements on short notice. Hence, the Linux kernel exhibits mechanisms that allow for tailoring it to its specific purpose, and thus, essentially forms a *software program family* [50, 10, 3] (a. k. a., variant-rich software system).

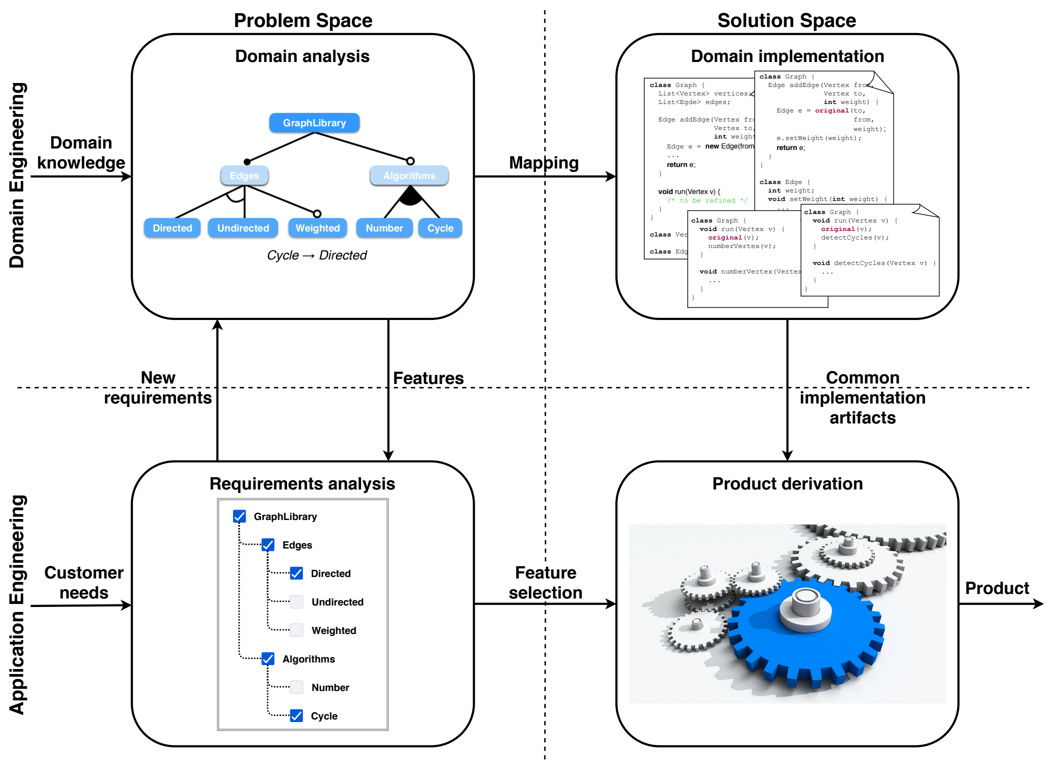
The core concept to achieve this variability are *configuration options* that allow to add or remove parts of the variant-rich software system when creating a particular variant. For Linux, these configuration options exist on different levels. In particular, Linux makes use of the *C preprocessor*, a configuration mechanisms that comes with the C programming language [26], for configuration on source-code level. Moreover, Linux has a dedicated language, Kbuild to allow for configuration on build file level.

While the benefits, such as reduced costs, shorter time to market, or increased reliability, of developing a program family are indisputable, they can not be obtained for free. Instead, a certain reuse process must be established to manage all these variants under one umbrella. One development paradigm that has been specifically proposed for variant-rich software systems is SPLE, which supports the development of "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [9]. In other words, SPLE fosters development and evolution of artifacts *across* all variants with only a minimum of manual effort necessary to create a new variant out of these artifacts.

To this end, SPLE structures the development process in two, orthogonal dimensions (cf. Figure 2.1). First, *Domain Engineering (DE)* and *Application Engineering (AE)* constitute one dimension, where DE is mainly concerned about the development (process) of *domain artifacts*, that is, artifacts that are reusable across all possible variants of the software system. In contrast, AE is concerned about the creation of concrete variant (a. k. a., *configuration*). This includes the specification of requirements, the selection of appropriate artifacts, their adaptation (if needed), and finally, their composition into a deployable software product.

As a second dimension that is orthogonal to the first one, *problem* and *solution space* are introduced by SPLE. The former covers all aspect of managing domain knowledge, such as defining the scope of a variant-rich software system (called *domain analsyis)* while the latter covers application-specific aspects, such as specifying requirements (and how these are covered on domain level) for a concrete variant.

Beyond the different dimensions, the notion of a *feature* is imperative, specifically to the SPLE process as well as to the development of variant-rich software systems in general. In this context, a feature is defined as a characteristics or user-visible behavior of a variant-rich software system ([3], p. 22). As such, it not only allows to reason about domain-specific aspects, but also to distinguish between variants, that is, there commonalities and differences in terms of features. As part of the domain analysis, the scope is defined in terms of features and the relations between them, usually resulting in



```
class Graph {
  List<Vertex> vertices;
  List<Egde> edges;

  Edge addEdge(Vertex fro
               Vertex to,
               int weight
    Edge e = new Edge(from
    ...
    return e;
  }

  void run(Vertex v) {
    /* to be refined */
  }
}

class Ver

class Edg
```

```
class Graph {
  Edge addEdge(Vertex from,
               Vertex to,
               int weight) {
    Edge e = original(to,
                      from,
                      weight);
    e.setWeight(weight);
    return e;
  }
}

class Edge {
  int weight;
  void setWeight(int weight) {
    ...
```

```
class Graph {
  void run(Vertex v) {
    original(v);
    numberVertex(v);
  }

  void numberVertex(Vertex
    ...
  }
}
```

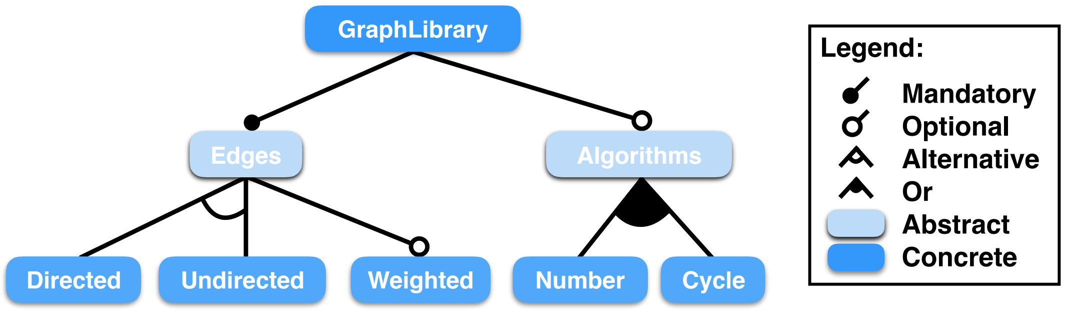
```
class Graph {
  void run(Vertex v) {
    original(v);
    detectCycles(v);
  }

  void detectCycles(Vertex v) {
    ...
  }
}
```

Figure 2.1: Overview of SPLE process (based on Apel et al. [3])

a *variability model.* The most popular form of such a variability model is a *feature model (FM)* which come with a graphical representation called *feature diagram.* In Figure 2.2, I show an example of such a feature model for the GraphPL product line.[2] In a nutshell, such a model specifies which features should be in every variant (*mandatory*) or can be included optionally. Moreover, relations between groups of features are specified, such as, whether features are mutually exclusive (*alternative* group) or can be selected together (*OR* group).

Given a variability model on domain level, a corresponding mechanism is needed on implementation level, together with a *mapping* between features (on domain level) and any existing implementation artifact [7]. Otherwise, the selection of features, e. g., for deriving a variant, can not be propagated to the solution space. As a result, these artifacts are not reusable as it would be impossible to tailor them according to the requirements in the product derivation process. Thus, to specify variable parts in artifacts, *variability (implementation) mechanisms* have been proposed that allow to map the features of domain level to the implementation level. Among others, preprocessors, FOP, or DOP are mechanisms that are commonly used [3]. I will introduce these mechanisms in more detail in the next section.

[2]http://spl2go.cs.ovgu.de/projects/49



Cycle → *Directed*

Figure 2.2: Feature model with feature diagram of the GRAPHPL (based on Apel et al. [3])

To illustrate how SPLE is applied in practice, let's go back to the LINUX example. To model variability on domain level, LINUX comes with two dedicated languages, KCONFIG and the already mentioned KBUILD. Both allow to specify features and their relations (e. g., alternatives, exclude, or required relations), but differ in the artifact they are mapped to on implementation level. While KCONFIG defines configuration options that are later used in the source code, KBUILD is mapped to the build level, that is, it specifies variability among modules. Nevertheless, both tools together have been commonly used to extract a comprehensive variability model for LINUX [8, 61]. On implementation level, LINUX employs the built-in C preprocessor that comes with the C programming language, which makes use of the configuration options in the KCONFIG files. To create a specific variant, a *configuration file* is used where for each feature it is specified whether it should be part of the configuration or not. This configuration file is then used to select and compose the tailored artifacts for product derivation.

Finally, it is worth to note that the development of variant-rich system with structured reuse does not always strictly adhere to the introduced SPLE process. Instead, it may vary by modifying or even omitting parts of the process. For instance, for many systems, an explicit variability model does not exist, which indicates the absence of a domain analysis. Nevertheless, these systems come with variability implementation mechanisms, and thus, allow to derive multiple variants of the system.

2.2 Integrated Variability: C Preprocessor and Feature-Oriented Programming

As mentioned in the previous section, different variability implementation mechanisms are available to realize integrated variability on implementation level. While some of them are inherently supported of almost all languages, such as parameters or design patterns (Apel et al. [3], Chapter 4), the focus of this thesis is more on advanced language-based and tool based variability mechanisms (Apel et al. [3], Chapter 5 & 6). Generally, these mechanisms can be classified into two categories: *annotation-based* and *composition-based* variability [24, 3]. Next, I will introduce two representatives for these categories, The C preprocessor (CPP) and FOP and how to apply them on source code level. The reason is that I consider on these mechanisms as well as sour code as development artifact for the research on integrated variability, summarized in this thesis.

The C Preprocessor

The annotation-based variability mechanism generally follows the *annotate-and-remove* paradigm: Initially, parts of the source code are annotated with their corresponding configuration option (i. e., the feature they belong to). Afterwards, these annotations can be employed to remove or keep the annotated part for the final derivation of a variant, depending on the features selected. As a result, all features are implemented in a single code base and in a non-modular way, thus, constituting a *virtual separation of concerns* [24, 3].

```c
#include <stdio.h>
int main(int argc, char **argv) {
#if defined(GUESS_POS) || defined(GUESS_NEG)
  int x;
#endif
  printf("Hello world!\n");
#if defined(GUESS_POS) || defined(GUESS_NEG)
  printf("What is my favorite number? ");
  scanf("%d", &x);
#ifdef GUESS_POS
  printf("Yes, %d is my favorite number!\n", x);
#else
  printf("No, %d is my favorite number!\n", x+1);
#endif
#endif
  return 0;
}
```

Listing 2.1: A "Hello world" example for integrated variability with C preprocessor (CPP) directives.

A prominent and widely used tool for annotation-based variability is the *C preprocessor (CPP)* that is integrated with the C programming language since its very beginning [26]. As such, the CPP provides lightweight metaprogramming capabilities, in particular, for macro definition (using `#define` directives), file inclusion (using `#include` directives), and conditional compilation (using preprocessor directives such as `#ifdef`, `#ifndef`, `#elif`, etc.). Especially the latter, also referred to as *preprocessor annotation*, is commonly used to introduce variability on code level in a fine-grained way, as the CPP allows to wrap around code elements even on expression or statement level to introduce conditional compilation.

In Listing 2.1, I show an example for CPP usage by means of a simple `HelloWorld` program, encompassing three different variants. First, configuration options actually constitute macros on code level. These macros form a *boolean expression* that is part of the preprocessor directive. For instance, in the example on Line 3, an `#if` directive introduces a boolean expression with two macros, concatenated with a logical operator: `GUESS_POS` and `GUESS_NEG`. The directive ends on Line 5, and thus, has only an effect on Line 4. In a nutshell, Line 4 is only part of the final program if either `GUESS_POS` or `GUESS_NEG` are defined (i. e., if at least on of these macros is `true`). Otherwise, the statement in Line 4 is removed in a preprocessing step that takes place *before* compilation. In the remainder, I will refer to such macros that are part of preprocessor directives as *preprocessor variables*.

Besides defining macros, a preprocessor directive can also rely on already defined macros (cf. Line 10), provide alternatives if a preprocessor directive resolves to false (cf. Lines 10–14), and even allows for nested preprocessor directives (cf. Line 7 and Line 10).

The common adoption of preprocessor-based variability (and the CPP in particular) is rooted by its beneficial properties. First, the concept of preprocessors is well known to developers, even without knowledge of variability. Moreover, the CPP is independent from its host language, and thus, similar mechanisms exist for a variety of languages (Apel et al. [3], Chapter 5.3.3). Second, the CPP easy and straightforward to use, because of the *annotate-and-remove* model, but still provides a considerable expressiveness. Most notably, it is very flexible as a developer can annotate even single characters but also large portions of code; the CPP can deal with all levels of granularity [25]. Finally, the CPP scales up even to large code bases. An example of its scalability is the LINUX kernel, where the CPP is used on millions lines of code to introduce variability for around 15 000 configuration options.

Feature-Oriented Programming

FOP is a composition-based variability mechanism that extends classical programming languages by an explicit notion of features. As initially introduced, FOP aims at decomposing a system into features, thus, making them first-class entities [53]. To this end, source code is organized in *feature modules*, which are orthogonal to (object-oriented) classes and capture all aspects of a particular feature by following ideas of *collaboration-based design* [66, 62]. As a result, FOP not only enables a *physical separation of concerns*; it also allows for a straightforward mapping between features (from problem space) and feature modules as well as an easy composition of such modules. Finally, due to its inherent modularity, FOP gives rise to modular reasoning.

In Listing 2.2, I show an example for an FOP implementation, of the GRAPHPL product lines using FEATUREHOUSEas a concrete composition technique [4]. This excerpt comprises four features and each feature comprises one or more classes. As a first fundamental difference to common object-oriented programming, a certain class can exist more than once in the whole program. In fact, a class exist for each role it plays in a certain feature. For instance, the class `Graph` has a role in each of the features, and thus, exists for times. Each feature, in turn, can extend the program in two ways: Either, *Feature* GraphLibrary

```
class Graph {
  List<Vertex> vertices;
  List<Egde> edges;

  Edge addEdge(Vertex from, Vertex to,
               int weight) {
    Edge e = new Edge(from, to);
    ...
    return e;
  }

  void run(Vertex v) {
    /* to be refined by Algorithms */
  }
}
```

```
class Vertex { ... }
```

```
class Edge { ... }
```

(a)

Feature Weighted

```
class Graph {
  Edge addEdge(Vertex from, Vertex to,
               int weight) {
    Edge e = original(to, from, weight);
    e.setWeight(weight);
    return e;
  }
}
```

```
class Edge {
  int weight;
  void setWeight(int weight) {...}
}
```

(b)

Feature Number

```
class Graph {
  void run(Vertex v) {
    original(v);
    numberVertex(v);
  }

  void numberVertex(Vertex v) {...}
}
```

(c)

Feature Cycle

```
class Graph {
  void run(Vertex v) {
    original(v);
    detectCycles(v);
  }

  void detectCycles(Vertex v) {...}
}
```

(d)

Listing 2.2: Excerpt of the GraphPL implementation using FeatureHouse

it *introduces* new elements (such as methods or fields) or it *refines* existing ones. In the example, feature *GraphLibrary* (Listing 2.2 (a)) constitutes the root feature, and thus, only introduces elements, such as the methods `addEdge(...)` and `run(...)` in class `Graph`. The other features rely on these introductions and partially extend them using the `original` keyword. For instance features *Number* (Listing 2.2 (c)) and *Cycle* (Listing 2.2 (d)) both refine the method `run(...)` by adding additional statements to it, while feature *Weighted* (Listing 2.2 (b)) extends method `addEdge(...)` by statements that add weights to edges. Note that the `original` keyword works similar as the `super` keyword with inheritance with the difference that the same class is extended, but within another feature module. Eventually, when the final program is composed based on the selected features, all elements across feature modules that belong together (e. g., classes, methods, etc.) are composed into one unique unit.

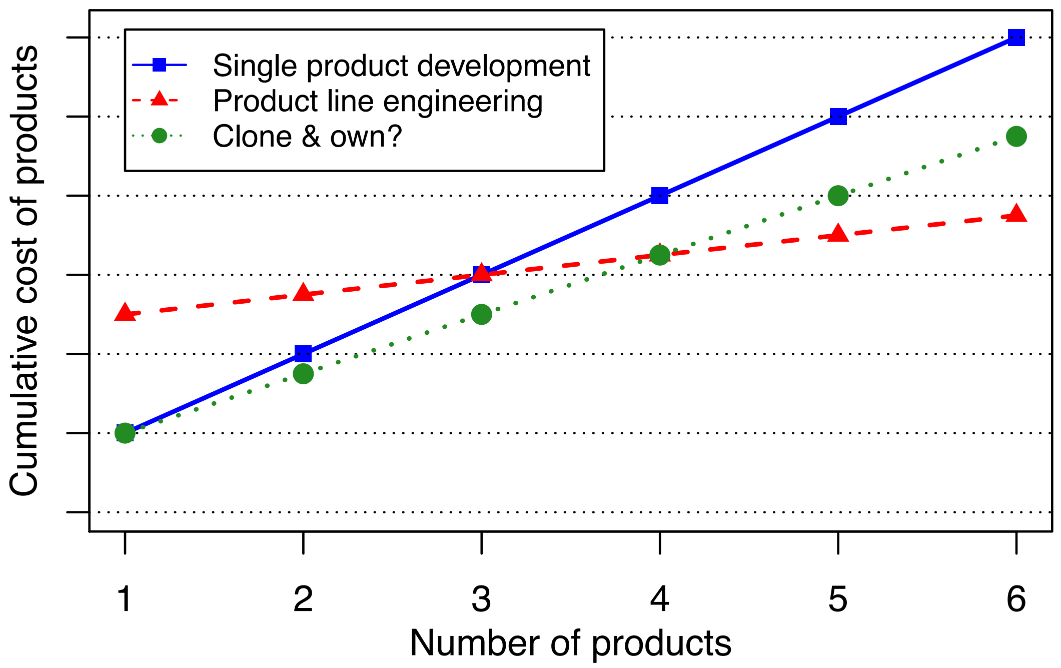


Figure 2.3: Comparing costs for SSS, SPLE. and *clone-and-own* development with respect to the number of products/variants expected (based on Clements and Northrop [9])

2.3 Adhoc Reuse with Clone-and-Own

While SPLE provides a way for structured and planned reuse when creating variant-rich system, and thus, to ease the evolution of thousand of variants in parallel, it also comes with high upfront costs (cf. Figure 2.3) [32, 52, 11]. In particular, the domain analysis is a tedious task and also maintaining a consistent mapping between features and artifacts comes not for free [9, 42]. Moreover, when starting a new software system from scratch, it is usually not known whether and how many variants will be needed [30]. Hence, most of the time, a development project starts with one singe system in mind that is developed. Later, with increasing demand of specific requirements, also the need for efficient creation of multiple variants arise. However, accomplishing the transition from an originally SSS to integrated variability is also a tedious and risky task [32].

Hence, as an alternative to SPLE, adhoc reuse is often used in practice, mainly by means of *clone-and-own* [14, 11, 56, 65]. This adhoc reuse comes with several advantages.

1. No upfront costs incur, making adhoc reuse a quick and cost-effective method to create variant-rich systems. Moreover, deriving a new system from an existing one does not affect the stability of the original one [23, 65].
2. Clone-and-own allows to reuse already established and tested code, and thus, increases the reliability and stability of the newly created variant [31].

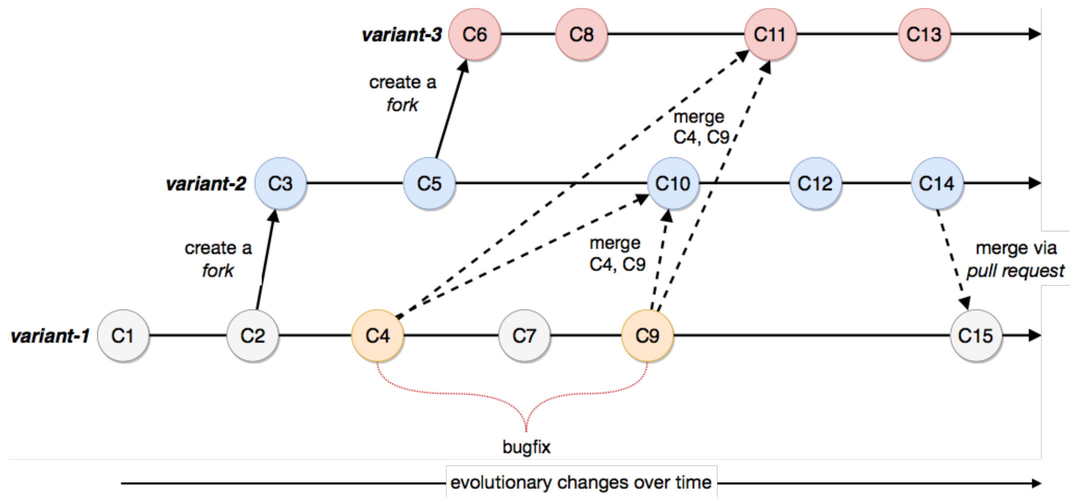


Figure 2.4: Creation and evolution of three variants by means of forking.

3. The concept of *clone-and-own* is very simple, and thus, can be applied straightforward without the need of complex tools or to adhere to formal processes. It is, in its most simplistic form, just copy-and-paste.

To create a variant with adhoc reuse, basically two ways exist: First, to physically *copy&paste* the respective files (or even the whole system) to another place. This creates an *informal fork* [65], that is, the newly created variant has no more relation or traceability link to the original system. Second, the built-in capabilities of modern *Version Control Systems (VCSs)*, commonly known as *forking*, can be used to derive a new variant based on an existing software system. In Figure 2.4, I illustrate the main steps and characteristics for applying forking by means of three variants and the GIT VCS.[3]

Creating a variant (a. k. a., forking:) As a preliminary, note that with GIT, you have always two repositories: A *local repository* that resides on your local computer; and a *remtote repository* that resides on a server, and thus, is available for everybody who has access to this server. Now, creating a fork means that the remote repository is copied, and thus, creates a clone of the original system on the server. For instance, on GITHUB a dedicated button exists that fulfills this tasks. As a result, a copy of the forked system will be created under the account of the user who creates the fork (i. e., presses the button). IN the example in Figure 2.4, *variant-2* is a fork of *variant-1* and *variant-3* is a fork of *variant-2*. This also illustrates that the process of *forking can be applied recursively*.

Evolving a variant: Now, after the fork has been created the user can clone this repository to her local machine (using `git clone`) and change whatever she wants. Moreover, synchronizing the local repository with the *own* remote repository can be done using `git push` and `git pull`, respectively. For instance, in the example *variant-2* is evolved by 5 *commits*, that is, changes that have been stored in the

[3]https://git-scm.com/

local repository (and which can be pushed to the remote one, if wanted). However, there is one important aspect I want to highlight: Each variant is evolved *independently*, that is, changes of one variant are not shared with the other variants.

Synchronization between variants: One advantage of adhoc reuse with forking is that, although each variant is evolved independently, it is possible to synchronize changes between variants. This is of superior importance in case of new features to be reused across variants or to propagate bug fixes. How this synchronization takes place depends on the direction, i. e., whether we propagate changes form the original system to the fork, or vice versa. For the former, consider *variant-1* and *variant-2* in our example. The developer of *variant-2* wants to take over bug fixes from *variant-2*. Since a forking relation exists, this can be simply accomplished by performain a *pull from upstream*, that is, by just merging changes from the original system into the fork. For the other way, the forked variant has usually no sufficient rights to make changes on the original repository, and thus, can not simply push changes to the original system (e. g., commit C14 in *variant-2* of the example). However, it is possible to file a *pull request*, which is kind of a proposal for a change. This will be checked by developers of the original system and, in case of acceptance, merged into this system (e. g., commit C15 of *variant-1* in the example).

In summary, compared to simple cpy&paste, adhoc reuse with forking has the advantages that it creates a *traceability link* between the original system and the new variant. This link can be used for future synchronization, but requires additional effort, as I will highlight in the next section.

2.4 Challenges for Evolving Variant-Rich Systems

Reuse is crucial to support the development, maintenance, and evolution of long-living software systems. This is even more true for variant-rich systems, which comprise a possibly large number of variants that must adhere to high quality standards at any stage of their lifecycle. I introduced and explained two different approaches, structured and adhoc reuse, that support the construction of variant-rich systems and partially even their evolution (which is especially true for the former).

However, these approaches are not for free. Especially when realizing these approaches with concrete techniques (e. g., integrated variability mechanisms or VCS), also disadvantages show up that inevitably pose challenges to the evolution of variant-rich systems with respect to quality aspects such as reliability, extensibility, maintainability, or testability. In the following, I will highlight the main challenges for evolution that arise from both approaches, with a main focus on the implementation level (i. e., solution space in the SPLE process).

2.4.1 Challenges of Integrated Variability

For integrated variability, I mainly focus on the preprocessor-based implementation mechanism with CPP, as it is (A) more commonly used in practice and (B) my research for integrated variability is mainly focussing on this mechanism. However, for the scattering aspect, I also take FOP into account.

The following *integrated variability challenges (IVC)* have been identified for the CPP and are addressed by parts of my research:

IVC 1 **Program Comprehension.** Since the CPP is part of the source code it is considered to be *intrusive*, tangling up with the host language. As a result, the CPP is criticized to obfuscate the source code of the underlying program [64, 40]. Moreover, developers are confronted with multiple variants at once and they have to understand and modify this code base on a regular basis. Understanding and working on a large number of variants simultanuously has been shown to negatively affect the comprehension of source code [47]. Finally, the intrusiveness of the CPP makes it inherently hard to follow and understand the data and control flow of the actual program [64, 15, 13]. In particular, the capability of introducing variability at any granularity leads to *undisciplined* annotations [39] that impede program comprehension considerably [43, 41, 45].

IVC 2 **Scattering and Tangling.** As features are not modularized with preprocessor-based variability, they can literally occur everywhere in the source code, thus, giving rise to *feature scattering*. This, in turn,makes it hard to understand which code contributes to a particular feature, as is may be scattered over hundreds of files [24]. Additionally, a feature rarely occurs in isolation. Instead, it may be *tangled* up with other features, such as in complex preprocessor expressions (cf. Listing 2.1, Line 3) or due to nested preprocessor directives. Both, scattering and tangling, hinder traceability of a features and modular reasoning [64, 13, 5]. Scattering also exists for FOP, although it supports a separation of concerns. However, the problem of scattering arises in case of too many refinements, e. g., for a particular method In such a case, the method may exist in many variants, depending on the selected features. As a result, scattering makes it hard to understand which previous refinements have to be taken into account when extending this methods as part of a new feature.

IVC 3 **Error-Proneness.** As a consequence of its expressiveness and the possibility of annotate code even at fine-grain, the CPP is also known to be prone to subtle errors, especially syntax and type errors [13, 44, 46, 49]. This leads to the occurrence of so-called *variability bugs* [1], which are hard to detect since there is only limited tool support to analyze variant-rich systems with preprocessor annotations.

IVC 4 **Maintainability.** This challenge is rather a consequence of the challenges mentioned above. Obviously, if code is hard to understand, impedes traceability and is prone to introduce subtle errors, it eventually also impairs the maintainability of variant-rich software systems, such as fixing a certain bug, modifying code according to changed requirements, or even extend it by new features.

While there are further challenges, such as testing all variants or perform static analyses on variant-rich systems, they are out of scope of this thesis. In summary, all of the above mentioned challenges makes it a non-trivial task to evolve a variant-rich software system properly. To address parts of these challenges, I will introduce respective techniques and empirical studies in Chapter 3.

2.4.2 Challenges of Clone-and-Own

Different to integrated variability, *clone-and-own* does not employ any specific variability mechanism. Instead, adhoc reuse is preferred, mainly by means of using the forking mechanisms of contemporary version control systems such as Git. Next, I highlight the challenges for evolving variant-rich systems developed with such adhoc reuse. To this end, I not only rely on previous research, but also make use of information that we obtained from an empirical study that we conducted on an open-source 3D printer firmware called Marlin [65]. In this study, we analyzed the repository of Marlin, how and why forking has been applied, qualitatively analyzed the shortcomings of this approach and also asked developers about problems arising from *clone-and-own*. As a result, the following *clone-and-own challenges (COC)* have been identified that are the main driver for my research on extracting variability (cf. Chapter 4):

COC 1 **Decentralized (Variability) Information.** One of the most intricate problems of *clone-and-own* is that information about commonalities and differences is simply not available. Even with modern VCS, this information is not stored persistently, and thus, makes it hard what is still identical and what has been changed between different variants. While this information may be easily obtained for a pair of variants, it is impossible to recreate this information for hundreds of forks, as we observed them in Marlin. A a result, reuse of features, implemented in a certain fork, is almost impossblie just because of this missing knowledge. In Marlin, we found out that there are more than 300 forks that developed new features and that some of them basically developed a feature redundantly (i. e., the same feature has been developed several times). Hence, the big picture about which features exists and how they may depend on each other is not available [6], which hinders their propagation/exchange across variants.

COC 2 **Maintenance Effort.** Similar to code clones [22], adhoc reuse for variant-rich systems suffers form an increased maintenance effort [14]. The reason is that any changes that are made in a particular fork (i. e., variant), are by default *not visible* to all other variants, including the original one. Again, a missing central knowledge base is the main reason for this issue. Instead, the owner of a fork must prepare *pull request (PR)* to the variant the fork has been created from. This may be a tedious task, as PRs are only accepted if they are robust enough, e. g., demonstrated by passing test suites. This may be even more complicated if the original system also evolved, as it may lead to conflicts or inconsistencies with modified or newly introduced features. Hence, it is common that the same changes are done manually over and over again on different forks, which binds resources for a task that could be mostly automated.

COC 3 **Change Propagation.** This challenge is related to COC 2, but has its own specialities. In particular, in case of bug fixes, it is of superior interest to propagate them to all related variants, that is, variants that comprise the same feature. and thus, suffer from this bug. However, this does not work consistently in both directions. First, if a bug is fixed by means of a *patch* in the original system, all direct forks just have to synchronize with this repository. However, either because of missing knowledge (cf. COC 1) or because the sync would require further adaptions (due to previous evolution of the fork), forks may not take this patch, and thus, still exhibit the bug. Even worse, by forks created form this (buggy) fork, this bug may even propagate further. In our study on Marlin, we found particular bug fixes (preventing the printer form being damaged), that have been taken by only $\sim 8\%$ of the forks. Second, forks that fix bugs do not necessarily push these changes back, for the reasons stated in COC 2, and thus, do not even provide other forks with the possibility to take this patch.

COC 4 **Difficult to Migrate.** Due to the lack of explicit feature and variability information, it is difficult to migrate form adhoc reuse to, either to an integrated variability mechanism or to an intermediate solution such as an integrated platform [2]. Hence, even if variants accumulate and such a migration would be beneficial, also from an economic point of view, the obstacle is mostly too huge to overcome it with reasonable effort.

With my research, I aim at overcome one of the root causes for the abovementioned challenges: I develop techniques that allow to extract information about features and their variability from multiple variants, developed with *clone-and-own* (see Chapter 4). Moreover, these techniques focus on different development artifacts such as models, code, or requirements.

3. Analyzing the Impact of Preprocessor Directives on Source Code Quality

In this chapter, I summarize techniques, together with their underlying concepts and empirical evaluation, to identify patterns of bad use of integrated variability and how this affects quality aspects such as understandability, maintainability, and evolvability. The particular contributions include (A) a systematic derivation of *variability-aware code smells (short: variability smells)*, based on well-established code smells (B) an empirical evaluation of the existence of such smells and its impact on quality aspects by means of a survey (C) a scalable and parameterizable technique, and its empirical evaluation, to detect such variability smells automatically, and (D) a large-scale empirical study and statistical analysis on the effect of preprocessor-based variability on maintenance effort.

3.1 Investigating Variability-Aware Code Smells in-the-wild

Relevant Publications: *This chapter summarizes the work published in Fenske and Schulze [17], Fenske et al. [18], and Schulze and Fenske [60], respectively. The former is covered in Section 3.1.1 and comprises the notion of variability-aware code smells and its evaluation while the last two are covered in Section 3.1.2 and comprise the metric-based technique and tool to detect such smells automatically.*

3.1.1 A Variability Perspective on Code Smells

Although previous studies investigated aspects of integrated variability such as undisciplined annotations or tangling and scattering, these studies are limited in two ways.

First, they do not analyze and validate to what extent these aspects have an impact on source code quality such as maintainability or extensibility. Second, they are usually analyzed quantitatively (i. e., purely value or metric-based) and in isolation, that is, it is not considered how the occurrence of multiple of such aspects affect the underlying source code.

We overcome these limitations by relying on a well-established concept of the domain of single software systems: *code smells* have been proposed to indicate higher-level pattern of source code indicating that a system suffers from design flaws or code decay [20]. Initially, they have been defined to identify refactoring opportunities, and thus, recreate the "healthiness" of the system by removing such smells. For our purposes, one characteristic of such smells is of prior importance: While such smells can be even identified by (quantitative) metrics, it is usually the interaction of certain shortcomings (indicated by multiple metrics) that resemble constitute a particular code smell. Consequently, the notion of code smells inherently considers multiple source code properties together and not each of them in isolation.

Methodology. Since we focus on integrated variability mechanisms, our main idea is to take variability into account as a *first-class concepts* for the definition of smells, resulting into variability-aware code smells (called . To this end, we reviewed common existing code smells and discussed how variability mechanisms would affect typical language elements of the original smell definition. For instance, a common and widely observed code smell is LONG METHOD, indicating that a method is too long, and thus, suspicious to comprise to much functionality. We asked ourselves "How would this smell look like if many statements are annotated with `#fidefs`"? We followed this methodology for a couple of smells, in particular those that are observed quite commonly such as DUPLICATED CODE or LONG METHOD, as these are well-understood and also do not rely on object-oriented programming (which is crucial when considering the C preprocessor). Moreover, we applied out methodology for both variability mechanisms, that is, CPP and FOP.

Catalog of Variability Smells. Based on the previously described methodology, we identified three code smells for which we could derive variability smells. In Table 3.1, we provide an overview of these variability smells together with a brief description and a reference to the original code smell. To illustrate the concrete mechanics of such smells, the smell ANNOTATIONBUNDLE is explained in greater detail in the following.

In Listing 3.1, we show a function form MYSQL that exhibits the smell ANNOTATION-BUNDLE. Note that this smell not necessarily requires a function to be extraordinary long, but accumulates a large amount of variability. For instance, the function in the example only comprises 29 lines of code. However, the majority of the code is variable, that is, annotated with (different) preprocessor directives, which relates to feature scattering. Moreover, it contains multiple preprocessor variables and also complex pre-

Table 3.1: Overview of the derived variability-aware code smells (for details, see [17])

Original Smell	**cpp-based Variability Smell**	**FOP-based Variability Smell**
Duplicated Code	Inter-Feature Code that is duplicated across features. For instance, a code fragment that is duplicated, but annotated with different features, respectively.	Code Clones Code that is duplicated across features. For instance, a method that exists in identical or similar form, but in different feature modules.
Long Method	AnnotationBundle A method that contains many variable parts, controlled by a many different features. Indicated by many cpp directives that are even nested.	Long Refinement Chain A method with many variable parts caused by excessive feature refinement.
Long Parameter List & Speculative Generality	Latently Unused Parameter A method parameter is introduced by a feature that does not make use of it (i. e., it is not used in the method body). However, this parameter is used with another feature, extending this method (a. k. a., forward declaration).	Latently Unused Parameter see cpp-based smell description

processor expressions (cf. Line 6 and Line 22), indicating feature tangling. Finally, there are also nested preprocessor directives present (Lines 21–27).

As a result, understanding what this function is doing for only some of the features selected is difficult, which also makes it challenging to extend this function. This example also shows that our definition of variability smells captures different characteristics of preprocessor-based variability, and thus, captures the interaction of these characteristics rather than considering them in isolation, as done in previous studies.

Evaluation. Although we derived the variability smells with care, they may be subjective and only constitute our point of view. Hence, we can not be sure that our smells really capture problematic (mis)use of variability in source code. To address this issue, we conducted a survey among expert in the field to answer the following research questions.

RQ 1 Do our proposed smells exist in the design and implementation of SPLs?

RQ 2 Are our smells problematic with respect to different aspects of SPL development?

We designed the survey in the following way. First, we collected some meta data such as age, gender, or current position and responsibilities at work. Second, to ensure that all participants are eligible to take the survey, we measured their programming experience. To this end, we employed the programming questionnaire proposed by Feigenspan et

```c
sig_handler process_alarm(int sig __attribute__((unused)))
{
  sigset_t old_mask;

  if (thd_lib_detected == THD_LIB_LT && !pthread_equal(pthread_self(),
      alarm_thread)) {
#if defined(MAIN) && !defined(__bsdi__)
    printf("thread_alarm in process_alarm\n");
    fflush(stdout);
#endif
#ifdef SIGNAL_HANDLER_RESET_ON_DELIVERY
    my_sigset(thr_client_alarm, process_alarm);
#endif
    return;
  }

#ifndef USE_ALARM_THREAD
  pthread_sigmask(SIG_SETMASK, &full_signal_set, &old_mask);
  mysql_mutex_lock(&LOCK_alarm);
#endif
  process_alarm_part2(sig);
#ifndef USE_ALARM_THREAD
#if !defined(USE_ONE_SIGNAL_HAND) && defined(SIGNAL_HANDLER_RESET_ON_DELIVERY)
  my_sigset(THR_SERVER_ALARM, process_alarm);
#endif
  mysql_mutex_unlock(&LOCK_alarm);
  pthread_sigmask(SIG_SETMASK, &old_mask, NULL);
#endif
  return;
}
```

Listing 3.1: Example for the smell ANNOTATION BUNDLE, taken from MYSQL, version 5.6.17 (see also [17]).

al. [16]. Third, we asked for prior knowledge about code smells, as this may affect to what extent participants understand the notion of variability-aware code smells. Finally, we asked questions about the concrete variability smells, proposed by us. In particular, for each smell, we asked whether participants have observed them (and where) and how they would estimate its impact on program comprehension, maintainability, and evolvability.

Finally, we sent out the survey to participants of the FOSD meeting 2014[1], two weeks before the meeting took place. We received 17 answers from which two have been discarded due to incompleteness.

Results. In Figure 3.1, we show the results for RQ 1, that is, to what extent participants of the survey can confirm the occurrence of our proposed variability smells.

[1]www.fosd.de/meeting2014)

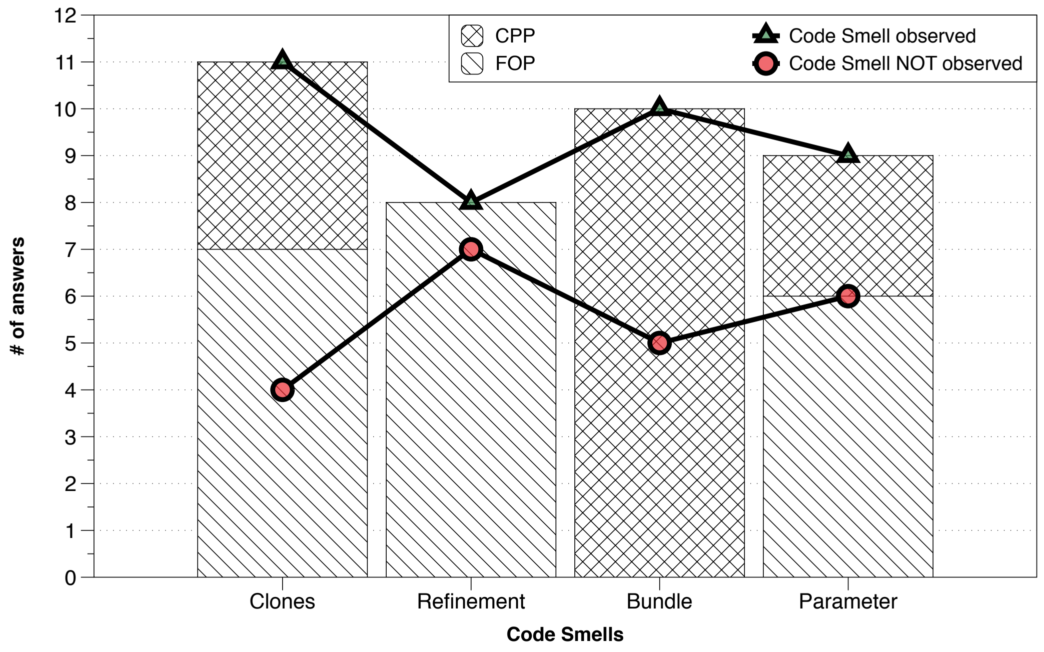


Figure 3.1: Occurrence of variability-aware code smells, as observed by survey participants.

As our data reveal, *all* smells have been observed by the majority of the survey participants, disregarding the variability mechanism. In particular, the smells Inter-Feature Code Clones and AnnotationBundle have been observed by $\sim 70\%$ of the participants. We argue that this result confirms that our proposed smells frequently exist in variant-rich software systems.

For RQ 2, we provide an overview of the results in Figure 3.2. The the charts indicate, there are differences between the variability smells and their impact on particular quality aspects. For instance, the smell Inter-Feature Code Clones has almost no impact on program comprehension, but is considered problematic for maintainability

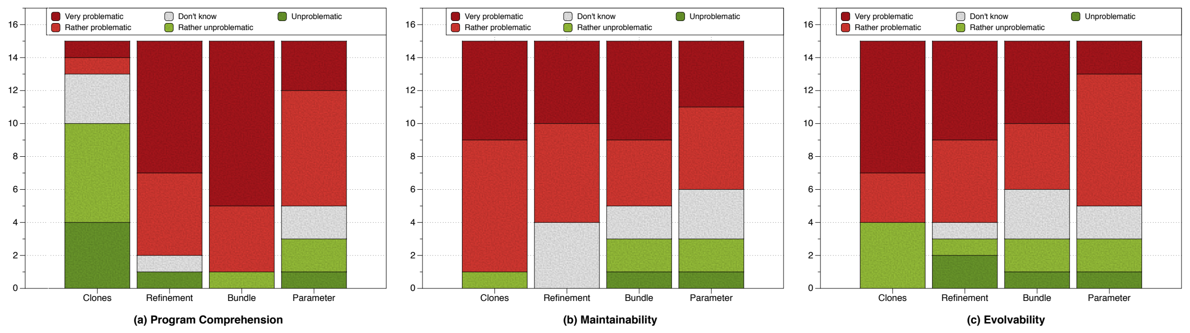


Figure 3.2: Impact of our variability-aware code smells on program comprehension, maintainability, and evolvability, respectively.

by almost all participants. This result is consistent with the nature of this smell, as usually a coherent piece of code is duplicated, which does not hinder understandability but increases maintenance effort. In contrast, the smell AnnotationBundle is perceived as (very) problematic with respect to program comprehension by almost all participants. Overall, with some minor exceptions, all variability smells are considered at least problematic by the majority participants for all three of the considered quality aspects. While this result was surprising in its clearness, it indicates that there is a need for identifying such variability-aware code smells and, if possible, to remove them.

3.1.2 Metric-Based Detection of Variability-Aware Code Smells

While our proposed variability-aware code smells have been confirmed by experts, these smells still rely on our own observations and are only available in a humn-readable description. However, to investigate the existence and nature of such clones at large scale, an automated detection process is needed. This is also necessary to guide developers and point them to possibly problematic code in large code bases.

Detection Technique. To detect variability smells, it is essential to find a roper technique or representation that captures the characteristic of such smells on a certain level of abstraction. As already explicated in Section 2.4.1 and Section 3.1.1, there are several characteristics that challenge the evolution of variant-rich system. How these characteristics contribute to a particular code smell and how to capture this interaction is the key point of providing a successful detection technique.

Eventually, we decided to employ a *metric-based* approach to detect variability-aware code smells, as metrics are suitable to capture particular characteristics of integrated variability [38, 39] and have also been used successful for detecting object-oriented code smells [12, 48]. Next, I illustrate how our technique works and which metrics we use by means of the smell AnnotationBundle (Note: corresponding metrics exist for the other smells as well). To this end, we make use of the code example in Listing 3.1, introduce the metrics used to describe this smell, and how we put these metrics into relation.

As already mentioned, the example in Listing 3.1 comprises multiple variable parts, complex preprocessor expressions, multiple configuration options, as well as nested preprocessor directives, all of them contributing the an AnnotationBundle. In Table 3.2, we provide the metrics that we propose to capture these characteristics together with a brief description. Except for the LOC metric, which basically counts all lines of code, all other metrics are specifically designed to address certain properties of cpp directive. As an example, the LOAC metric counts only the lines of annotated code, whereas the FL metric captures the number of variable parts (i. e., annotated blocks of code) within a function.

To put all of these metrics in relation, we propose to aggregate the particular values to one single metric, AB_{SMELL}, that indicates to what extent a function suffers from this smell. We show this metric in (Equation 3.1).

Table 3.2: Metrics capturing basic characteristics of the ANNOTATION BUNDLE code smell (adapted from [18])

Abbrev.	Full Name	Description
LOC	Lines of code	Source lines of code of the function, ignoring blank lines and comments.
LOAC	Lines of annotated code	Source lines of code in all feature locations within the function. Lines that occur in a nested feature location are counted only once. Again, blank lines and comments are ignored.
CND	Cumulative nesting depth	Nesting depth of annotations, accumulated over all feature locations within the scope. An `#ifdef` that is not enclosed by another `#ifdef` is called a *top-level* `#ifdef` and has a nesting depth of zero; an `#ifdef` within a top-level `#ifdef` has a nesting depth of one, and so on. Nesting values are accumulated, which means that a function containing two feature locations with a nesting depth of one is assigned a CND value of 2.
FC_{DUP}	Number of feature constants	Number of feature constants, accumulated over all feature locations within the scope. Feature constants that occur in multiple feature locations are counted multiple times.
FL	Number of feature locations	Number of blocks annotated with an `#ifdef`. An `#ifdef` containing a complex expression (e. g., `#ifdef A && B`) counts as a single feature location. An `#ifdef` with an `#else`/`#elif` branch counts as two locations.
NEG	Negation	The number of negations in the `#ifdef` directives in a function. Both `#ifndef X` and `#if !defined(X)` increase NEG by 1. `#else` branches also increase NEG because `#if <expr> ... #else ...` is treated as `#if <expr> ... #endif #if !<expr> ...`

$$w_1 \cdot \frac{LOAC}{LOC} \cdot FL + w_2 \cdot \frac{FC_{dup}}{FL} + w_3 \cdot \frac{CND}{FL} \tag{3.1}$$

The equation consists of three terms which capture the following characteristics. The first term mainly captures the amount of variable code, also taking into account how many variable parts exist in a function, and thus, accounting for scattering. Next, the second term addresses the number of preprocessor variables and how they are distributed over annotated code fragments. As a result, this term provides a way to relate scattering and tangling and integrate both of them into our smell definition as a proxy for complexity. As a third term, we also take the nesting depth into account, as this has been shown to affect the comprehension of the nested code fragments.

Finally, each term is complemented by a *weight*, which allows to control the influence that a particular term has on the overall metric. The reason is that for different developers or in different projects, the perception of what makes a function an ANNOTATION-BUNDLE is different. Consequently, introducing weights allows for *parameterization* of the actual metric-based definition of a smell. Besides the weights, we also provide customizable thresholds for each of the atomic metrics in Table 3.2. These thresholds

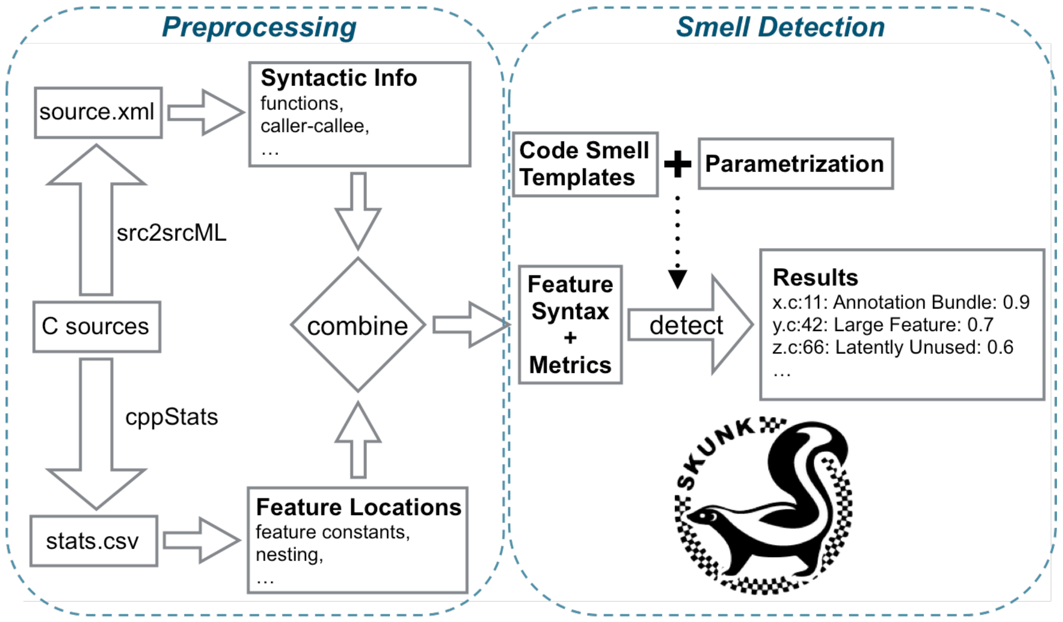


Figure 3.3: Overview of the (pipe&filter) architecture of Skunk, implementing the metric-based code smell detection.

constitute lower boundaries, that is, if the metric is below the threshold, it is not considered for the computation of the overall metric AB_{SMELL}. This way, we aim at reducing false positives in the result set.

Implementation. We implemented our technique in the tool Skunk[2] with a pipe&filter architecture so that any intermediate results are available for further usage. As shown in Figure 3.3, the tool consists of two stages: preprocessing and smell detection. In the preprocessing stage, the source code of the system of interest is analyzed. To this end, we employ to existing tools: cppstats[3] and srcML.[4] With cppstats, we extract most of the variability-related information that we need for our metrics, such as the location of preprocessor directives or the preprocessor variables involved. However, we need additional information, such as the location of functions definitions or information about function calls. This information we obtain from srcML, which provides a bootstrapped *Abstract Syntax Tree (AST)* in XML format. Most notably, srcML provides this AST with all preprocessor directives included, which is pivotal for any variability-aware analysis.

Based on the results of this preprocessing stage, Skunk performs the actual code smell detection. To this end, we initially extract all relevant information and compute our metrics, relevant for detecting a particular code smell (the **Feature Syntax + Metric**

[2]https://github.com/wfenske/Skunk
[3]http://www.fosd.net/cppstats/
[4]http://srcml.org/

part in Figure 3.3). Then, the actual smell detection process starts. To this end, we provide *code smell templates* for each of our proposed variability smells. This template not only specifies the metrics that are relevant for the smell to be detected; it also allows the user to specify the aforementioned thresholds for each of these metrics. For instance, a user can define the threshold for the LOAC/LOC ratio to be at least 50%. As a result, any function that does achieve this threshold is discarded from the further smell detection. Moreover, in this step the smell metric can be parameterized, that is, the weights can be adjusted. Finally, the code smell template and variability-related information is used to compute the smell metric (here: $\mathrm{AB}_{\mathrm{SMELL}}$) and the value is stored together with the function name and location. These results can now be investigated by the user to identify smelly functions and to initiate possible countermeasures.

Evaluation. To evaluate our technique, we conducted a case study on five open-source systems, among them long-living and popular systems such as VIM or EMACS. With this evaluation, we aim at answering the following research questions:

RQ 1 Does our algorithm detect meaningful instances of the AnnotationBundle smell?

RQ 2 Does the AnnotationBundle smell exhibit recurring, higher-level patterns of usage?

While RQ 1 is focussed more on the accuracy of our detection technique (quantitative analysis), RQ 2 is more about certain characteristics that contribute to the overall pattern (qualitative analysis). For the evaluation, we set up Skunk with a parameterized code smell template, based on our experiences with different variant-rich systems (cf. [18]). Then, we ran the tool on the five subject systems, all of them with a size of > 100 *KLOC* and an amount of variable code between 20% and 70%. As a result, we obtain a list for each system that contains the function, its location and the computed $\mathrm{AB}_{\mathrm{SMELL}}$ metric value (in descending order). Since there is no experience or baseline with variability-aware code smells and because our metric is not normailzed in a fixed range (e. g., between 0 and 1), it is difficult to determine, which metric values indicate a real smell (true positive) and which values can be neglected (false positives).

To overcome this uncertainty, we performed a manual analysis by sampling the results in the following way: First, for each system, we selected the 10 entries with the highest $\mathrm{AB}_{\mathrm{SMELL}}$ value. Second, we divided the remaining entries of the result list into 10 equally distributed segments and then randomly selected one entry per segment. As a result, we obtain 20 smells for each system, which have been manually analyzed by me and another author (cf. [18]). To this end, we independently evaluated each smell regarding its impact on understandability and maintainability on a three point scale ($\{-1, 0, 1\}$). Eventually, we compared our results and, in case of disagreeing for a particular smell, discussed their ratings to find a consensus.

In Figure 3.4, we show a summary of the quantitative results. Except for the subject system PHP, our results indicate that our definition for the AnnotationBundle

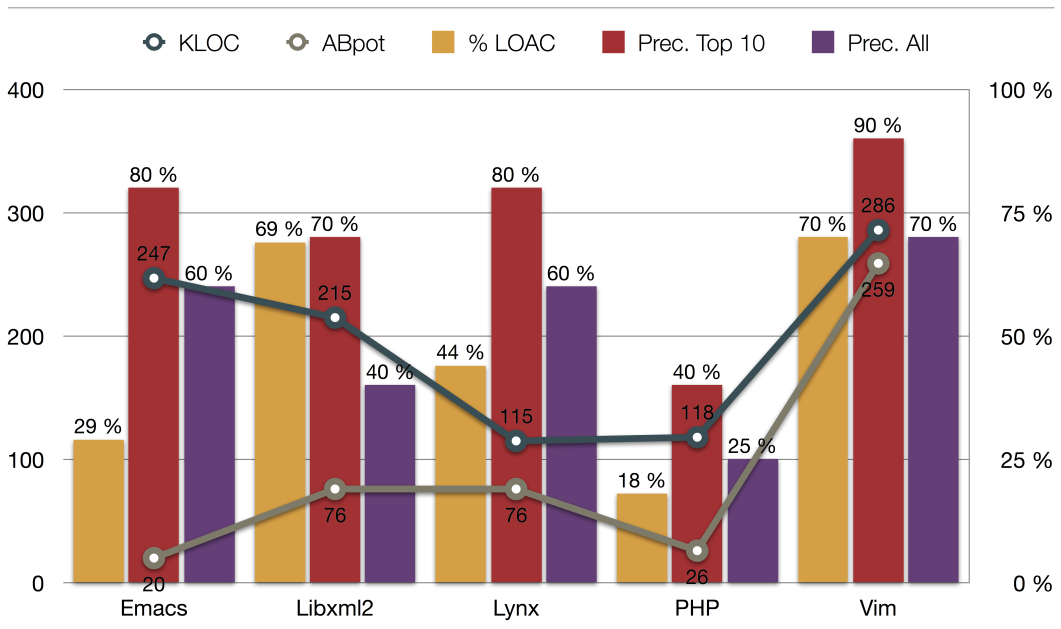


Figure 3.4: Overview of the quantitative results for the evaluation. The overall number of smells detected for each system is indicated by the value AB_{POT}.

smell can be considered appropriate and that our detection technique captures the characteristic of this small with high accuracy. Especially the top-10 entries of the result list have been shown to be very good indicators with an accuracy of more than 70%. Hence, we argue for RQ 1 that our technique is capable of detecting meaningful instances of the smell AnnotationBundle.

For our qualitative analysis, which relies on the manual inspection of the selected code smells (see above), we made several observations about the characteristics of the smell as well as for possible reasons why it has been introduced. Among others, we observed the following (more details about the observations can be round in [18]).

- There is no single reason, and thus, no single metric, that causes the AnnotationBundle smell. Rather, it is usually a combination of several characteristics that contribute to a smell.
- One particular aspect that fosters the occurrence of our smell is the interaction of preprocessor variability with runtime variability (i. e., conditional statements of the host language). The reason is that already complex control flow is obfuscated even more when annotated with preprocessor directives.
- Long functions are more prone to constitute a smell than short functions.
- A form of the *Adapter pattern* [21] is a recurring pattern that is likely to introduce the AnnotationBundle smell.
- Some functions constitute Featurized God Functions, a form of the God Class smell [20]. They are characterized not only by its length, but also by comprising many diverse features tangled up with each other and scattered across

the whole function. We observed this kind of functions especially in VIM and all suffered from the ANNOTATIONBUNDLE smell.

In summary, I conclude that the the smell ANNOTATIONBUNDLE is well captured by our detection technique and the metrics used; that it occurs quite frequently in systems from different domains; and that there are certain structural properties and even pattern that foster the occurrence of this smell.

3.2 How Preprocessor-Based Variability Affects Maintenance

Relevant Publication: *This section summarizes the content of the paper by Fenske et al. [19].*

With the work summarized above, we introduced the notion of variability and verified that these smells, in particular the ANNOTATIONBUNDLE frequently occurs in variant-rich software system. However, although experts in our survey confirmed that these smells are problematic regarding program comprehension or maintainability, so far no hard evidence exist that support this opinion. Hence, we conducted an empirical study to analyze the effect of preprocessor variability on maintainability.

Research Design. First, we need to find a measure for maintainability, that is, the maintenance effort for a particular function. In previous work, researchers proposed to use *change proneness* as a proxy for the maintenance effort and demonstrated the feasibility of this proxy [54, 27]. In particular, two measures are proposed: The *frequency of changes*, also known as an indicator for later defects. And the *amount of changes (a. k. a., code churn)*, know to correlate with the effort of developers for maintenance tasks. To this end, we introduce two *change metrics* for each function: the number of commits that have modified the function and the lines changed/removed, accumulated over a period of time. Moreover, we also compute a normalized version of these metrics, that is, dividing it by the length (in LOC) of the function. Since these metrics work on statement level (as smallest unit of change), we can't map them directly to our code smells (e. g., ANNOTATIONBUNDLE, because we aggregate fine-grained information of statement level up to function level. Hence, using the smell as a predictor for change proneness will leave us with too many false positives or false negatives. Hence, we decided to use the atomic metrics (cf. Table 3.2) to evaluate whether certain preprocessor usage has an effect on maintainability.

Next, we have to extract the relevant data for our analysis. While we can reuse our SKUNK infrastructure to extract variability-related information, it is only designed to do so for one version of a system. Hence, we developed a tool on top of SKUNK that not only computes our variability metrics for multiple versions, but also is capable to extract fine-grained changes from version control repositories at large scale. In a nutshell, our data extraction process works as follows:

1. We **identify relevant commits** in the repository, that is, commits in which at least one .c file has changed (compared to the previous commit). Changes to other files are irrelevant, and thus, ignored.
2. Next, we **create snapshots**, that is, a logical sequence of commits that are in a parent-child relationship. We do this, because due to different branches, e. g., for development or bug fixes, and merges of them to the main branch, the chronological order of commits is not necessarily the logical one. Obviously, this process may lead to snapshots of different size, which may later (accidentally) affect the precision of our analysis. Thus, we divide the initially created snapshots in equally sized ones, each comprising 100 commits. Note that we also compute our change and variability metrics only once per snapshot (see next step), as doing this for each and every commit would be computationally too expensive, especially when considering $\sim$ 30 years of development history.
3. Once we created our snapshots, we can **process them** to collect al necessary information for our analysis. To this end, for each snapshot, we compute variability metrics (using Skunk, perform change analysis and compute the change metrics, and, eventually, combine both sets of metrics. As a result, we obtain for each file a list of functions together with its variability and change metrics.
4. Finally, we **build commit windows**. The reason is that the size of a particular snapshot is too small to identify functions that really undergo heavy changes. Hence, we group 10 snapshots into one commit window and recompute the change metrics for each commit window.

Evaluation. For our evaluation, we formulate the following three research questions:

RQ 1 Is feature code harder to maintain than non-feature code?
RQ 2 Does the presence of feature code relate to the size of a piece of code?
RQ 3 Considering all properties of annotation usage and size in context, what is the independent effect of each property on maintainability?

For each question, we formulate null hypotheses and select appropriate statistical methods to verify these hypotheses and analyze whether preprocessor usage (by means of our variability metrics) correlates with maintainability (by means of our change metrics). Moreover, we selected eight subject systems for our evaluation, based on seven criteria. Most importantly, each system must have a sufficiently long history, and thus, a large amount of commits to perform our evolutionary analysis (i. e., change analysis).

For RQ 1, we show the correlation between our variability metrics and the measures for maintenance effort in the upper part of Table 3.3. For both maintainability measures (indicated as dependent variables COMMITS and LCHG, respectively), our data reveal a twofold result. First, the result indicate that the difference in maintainability is significant in the presence of variability[5] compared to non-variable code. This

[5]Given that the variable code fulfills the properties defined by our metrics (i. e., the minimum thresholds for each variability metric)

Table 3.3: Effect of Individual Annotation Metrics on Changes (taken from Fenske et al. [19])

Independent	Dependent	Sig.[1]	Cliff's Delta	Magnitude[2]		
$fl_{>0}$	COMMITS	8 (0)	0.27±0.11	○	○	◐
$fc_{>1}$	COMMITS	8 (0)	0.39±0.12	○	◐	●
$nd_{>0}$	COMMITS	8 (0)	0.40±0.14	○	◐	●
$neg_{>0}$	COMMITS	8 (0)	0.32±0.12	○	○	◐
$fl_{>0}$	LCHG	8 (0)	0.27±0.11	○	○	◐
$fc_{>1}$	LCHG	8 (0)	0.39±0.12	○	◐	●
$nd_{>0}$	LCHG	8 (0)	0.40±0.14	○	◐	●
$neg_{>0}$	LCHG	8 (0)	0.32±0.11	○	○	◐
loc^{+}	COMMITS	8 (0)	0.24±0.05	○	○	○
loc^{+}	LCHG	8 (0)	0.25±0.05	○	○	○
$fl_{>0}$	$\text{COMMITS}_{/\text{LOC}}$	7 (1)	0.22±0.07	○	○	○
$fc_{>1}$	$\text{COMMITS}_{/\text{LOC}}$	8 (0)	0.27±0.09	○	○	◐
$nd_{>0}$	$\text{COMMITS}_{/\text{LOC}}$	7 (1)	0.29±0.08	○	○	◐
$neg_{>0}$	$\text{COMMITS}_{/\text{LOC}}$	8 (0)	0.24±0.10	–	○	◐
$fl_{>0}$	$\text{LCHG}_{/\text{LOC}}$	8 (0)	0.22±0.10	–	○	○
$fc_{>1}$	$\text{LCHG}_{/\text{LOC}}$	8 (0)	0.31±0.10	○	○	◐
$nd_{>0}$	$\text{LCHG}_{/\text{LOC}}$	8 (0)	0.32±0.12	○	○	◐
$neg_{>0}$	$\text{LCHG}_{/\text{LOC}}$	8 (0)	0.27±0.10	○	○	◐

[1] Number of subject where the difference was significant at $p<0.01$ or not significant (in parentheses).

[2] Magnitude of d (Cliff's Delta), for $M(d)-SD(d)$, $M(d)$, and $M(d)+SD(d)$. –: negligible, ○: small, ◐: medium, ●: large

is confirmed by all subject systems, i. e., significance has been shown for all of them. Moreover, this significance exists for all variability metrics. Second, we can observe differences, regarding the particular variability metrics, in the effect size of the confirmed correlation. In particular, we could observe a medium positive correlation for the metrics FC and CND, whereas the metrics FL and NEG only exhibit a small positive correlation. Considering all subject systems and both maintainability measures in summary, the cumulated nesting depth (CND) stands out to have the largest effect. Overall, the results for RQ 1 let us conclude that functions with variable code (according to our metrics) are changed more frequently and to a greater extent than other functions.

For RQ 2, we take the *function size* (in terms of LOC) into account as well, because previous studies have shown that the size of the function also affects certain code properties, and thus, may consitute a *confounding factor* in our study. Consequently, we relate our variability metrics also to function size to checker whether a correlation exists. Indeed, our results indicate a positive correlation between function size and all variability metrics, that is, functions with more preprocessor annotations also tend to be longer. This raises the questions whether and to what extent the function size itself is a sufficient predictor for maintenance effort. To this end, we analyzed our maintainability metrics with function size as dependent variable (cf. Table 3.3, middle part). Our data reveal a positive effect of function size on both, change frequency and change amount of changes. Moreover, although with a small positive effect, all of these results are significant.

Since we could confirm, based on our evaluation, that function size is a confounding factor, we were interested strong the correlation of our variability metrics is when mitigating the effect of function size. To this end, we normalized both maintainability measures by function size and repeated the statistical analysis (cf. Table 3.3, lower part). Our data reveal the following results.

- For change frequency, while still being significant for most subject systems, the mean effect size for all variability metrics decreases (compared to RQ 1). While there are notable differences between subject systems (e. g., for GLIBC, all metrics but CND showed a negligible effect whereas for OpenVPN, two metrics showed a medium to large effect), on average we conclude that even with normalized function size, the effect of variability is still significant, but with a smaller effect size than without normalization.
- For the amount of changes, in contrast to change frequency, the effects for all subject systems remain significant (compared to RQ 1). However, even here we can observe a considerable decrease of the effect size, concluding that there is a visible effect of function size.

In summary, comparing the results of RQ 1 and RQ 2, we can observe a considerable difference in the effect size, mainly caused by functions size as a confounding factor. In particular, our results allow for the conclusion that function size has a major impact on change proneness. This, in turn, raises the question to what extent our variability metrics are well-suited as an predictor for maintainability.

We address this issue with RQ 3, where we evaluate the independent effect of each metric when we consider all of them together. To this end, we applied different *regression models* for all independent variables and show the results in Table 3.4. For both maintainability metrics, we investigate the independent effect of all our variability metrics as well as function size and show the results (A) for OpenVPN as a concrete subject system and (B) on average over all systems. Moreover, we compute different results for each regression model, indicated by columns β, $\mathbf{z}$, and $\mathbf{p}$. The coefficient β indicates to what extent the dependent variable is expected to increase given that the associated independent variable v_i is increased. For the z-score, the coefficient β together with the standard error is taken into account. Hence, high absolute values for $\mathbf{z}$ indicate that the independent variable predicts the dependent variable in a reliable manner.

For change frequency, our regression analysis reveals that for the metrics FL and NEG, no significant effects could be observed (indicated by the high p-value), and thus, no correlation with change frequency exists. While all other metrics (incl. function size) are significant, especially the variability metrics show only a rather small effect (indicated by the very small coefficients). For OpenVPN, CND even shows a slightly negative effect,

Table 3.4: Regression Models for answering RQ 3 (taken from Fenske et al. [19]).

	v_i	β	**z**	**p**	β	**z**	**Sig.**
		OpenLDAP			All Systems		
COMMITS	(Intercept)	-3.48	-120.5	<0.001	-3.16±0.44	-98.4±37.2	7
	FL	-0.00	-0.1	0.939	-0.11±0.06	-5.1±01.9	4
	FC	0.06	5.7	<0.001	0.20±0.19	7.3±01.5	6
	ND	-0.06	-7.8	<0.001	-0.07±0.02	-7.0±01.7	3
	NEG	0.02	1.9	0.058	-0.00±0.14	-1.1±09.9	3
	$\text{LOAC}_{/\text{LOC}}$	0.66	13.9	<0.001	0.44±0.25	7.5±04.9	6
	$\log_2(LOC)$	0.58	104.4	<0.001	0.54±0.05	79.9±29.4	7
LCHG	(Intercept)	-2.83	-69.9	<0.001	-2.71±0.43	-59.3±21.4	7
	FL	0.00	0.3	0.729	*na*	*na*	0
	FC	-0.04	-1.8	0.071	0.23±0.20	3.2±00.3	2
	ND	-0.04	-2.9	0.003	-0.06±0.02	-3.6±01.0	2
	NEG	0.02	1.1	0.268	0.02±0.16	0.6±05.5	2
	$\text{LOAC}_{/\text{LOC}}$	0.96	10.9	<0.001	0.69±0.30	6.4±03.6	5
	$\log_2(LOC)$	0.87	101.1	<0.001	0.87±0.04	83.5±29.4	7

β – coefficient estimate, **z** – z-value, **p** – p-value, **Sig.** – #systems with significant effects

meaning that if the nesting depths increases, the function is *less* likely to be changed. In contrast, we have a relatively large effect for the function size. For instance, for OpenVPN, the $\log_2(LOC)$ metric has β value of 0.58, which means that if a function doubles in size, the change frequency increases by 58%. The same tendency is indicated by the very high z-value of this metric. Hence, for change frequency, we can conclude that, while there is a small correlation with some variability metrics, function size is by far the most reliable predictor over all systems.

For the amount of changes, the results are similar yet different. In particular, the results differ in two ways. First, effect of our variability metrics is even less significant than for change frequency, and thus, close to be meaningless for predicting the dependent variable LCHG. Second, the effect of function size is only significant for five subject systems, but for those systems, the effect size is even greater than for change frequency.

Conclusion. Based on our results for the three research questions, which comprised established statistical methods, we make the following conclusions:

- Preprocessor annotations have an inconsistent effect on change proneness, that is, also we found the effect to be significant, the effect was varying with mostly being small or medium in size.
- Function size, as indicated in previous studies, has a more consistent effect, which is also significance, regarding the correlation with change proneness.
- Despite the two conclusions above, we argue that it is still a bad idea to create long functions with lots of variable code, especially because heavily annotated functions tend to be larger, which in turn increases the change proneness.

- Overall, our regression models had only a poor prediction accuracy, which indicates that other important factors are missing that have a more visible and sustainable impact on change proneness. Hence, in future work, we will investigate further metrics (e. g.., process metrics such as the age of code) to improve our models.

4. Analysis Techniques for Feature and Variability Extraction

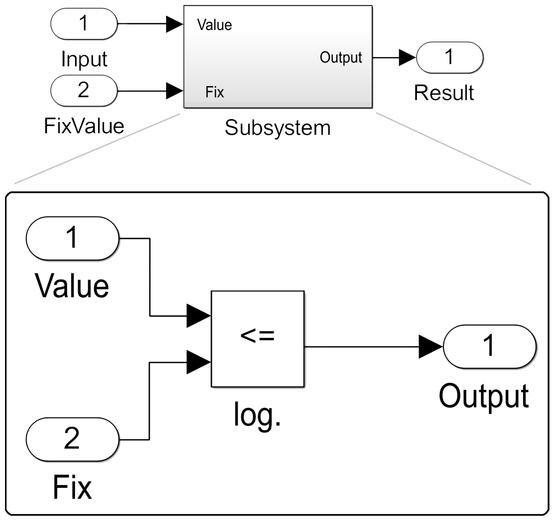
In this chapter, I summarize techniques and their empirical evaluation that enable the semi-automated extraction of features and variability (called *variability mining*) from different artifacts that usually accumulate in the software development process. In particular, we propose

- a model-based technique for variability mining of MATLAB/Simulink models with an evaluation of real-world models and a developer survey [59]. Moreover, I briefly report on a generlalization of this technique for arbitrary, block-based modeling languages [67].
- a static analysis techniques for identifying similarities across multiple MATLAB/Simulink models in an n-way fashion [58]. This technique is also a generalisation of pure variability mining as the results can also be used for other purposes.
- two techniques based on natural language processing and machine learning that allow to extract a feature model and (partially) their dependencies from natural language requirements [36, 35, 37].

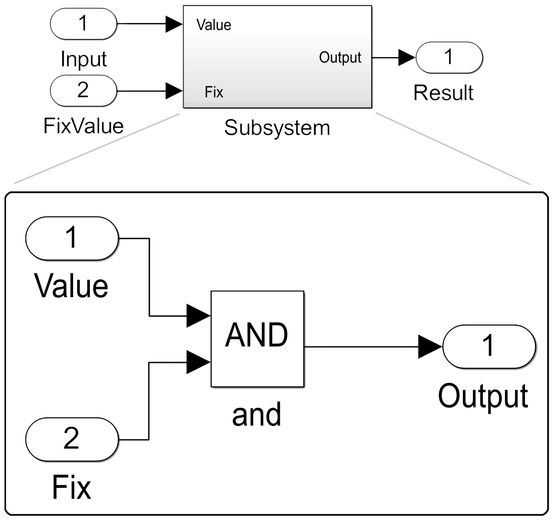
4.1 Extracting Variability from Block-Based Models

Model-based languages play an important role in domains, where the inherent complexity and cyber-physical aspects play an important role. For instance, in the automotive domain, the software for *Electronic Control Units (ECUs)* is mostly developed using MATLAB/Simulink models, as it allows for abstract away a certain complexity and to eventually generate the code out of these models.

In Figure 4.1, I show a pair of such MATLAB/Simulink models. With MATLAB/Simulink being a block-based, behavioral modeling language, the *function blocks* (e.g., *AND* in fig:matlab) are connected via *connectors* which are used to send *signals* between blocks. These signals can be considered as kind of data flow, thus, establishing dependencies between blocks. Moreover, each block can have in- and out-ports that constitute the interfaces of a block to send and receive data.



(a) Model Variant A



(b) Model Variant B

Figure 4.1: Two variants of a MATLAB/Simulink model with commonalities and differences.

Moreover, MATLAB/Simulink allows to introduce hierarchies in models. Especially when dealing with large models with thousands of blocks, this is extremely useful, as it allows to encompass parts of the model by one single block, thus, reducing the initial size of the model as details are hidden. In our example, both model variants have such a hierarchy, indicated by the *subsystem block*. This subsystem block is used in the highest modeling level, connected with two in-ports and one out-port. Within this block, there are *sub blocks* that constitute the actual functionality and are shifted to the next model layer. Hence, we could replace the subsystem by its content on the highest level, but then take into account that our model heavily increases.

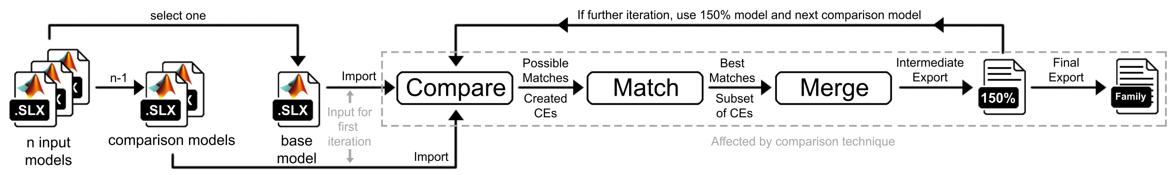


Figure 4.2: Overview of the workflow for variability mining of models.

Finally, both of our model variants in Figure 4.1 are very similar as they just differ by one function block in the subsystem. Such similar models are also quite common in practice. The reason is that, for large models, the effort is simply too high for creating models always from scratch. Instead, when a new model is needed (e. g., for a newly created ECU), another model that has a similar functionality is copied and than adapted according to the new requirements. This not only saves time, otherwise need to create thousands of blocks, but also increases the reliability, as the copied model has been already tested and applied in practice. On the other hand, having lots of similar models also comes with additional maintenance effort, especially if the same change has to be applied to all of them. Even worse, information about commonalities and differences between models is usually not documented, and thus, this information is not explicitly available.

The Family Mining Technique. To address the abovementioned problem and recreate the information, we proposed a technique variability mining between MATLAB/Simulink models, a. k. a., *family mining*. An overview of the workflow and particular processing steps is shown in Figure 4.2 and I will briefly explain each step in the following.

Initially, we have to define a *base model* out of our input model variants, for instance, the largest or smallest models. We then chose a another model from the remaining input models for the first comparison. Both, the base model and the comparison model, are then imported which basically mean that we transform then in an internal representation. To this end, we have defined a metamodel using the *Eclipse Modeling Framework (EMF)*. Hence, importing models basically constitutes a model-to-model transformation from the original MATLAB/Simulink model into a model that adheres to our metamodel. The reason is that we can now perform any further step based on elements of our metamodel, and thus, are independent of the actual modeling language of the input models.

Next, in the *compare phase*, we compare the particular blocks of our to input models. To this end, we have developed to approaches: the *data flow approach (DFA)* and, as an extension, a *matching window technique (MWT)*. DFA employs the fact that for each model, the elements of one hierarchy level can be divided into *stages*, according to the data flow between model blocks. For instance, in Figure 4.1 (a), the initial blocks *Input* and *FixValue* belong to one stage, where as the subsystem is in another stage due to the connector (i. e., the data flow separates the stages). Once we identified the stages in both compare models, we compare elements of one stage with each other. To logically associate compared blocks (for usage in later steps), we introduce a *Compare Element (CE)* in our metamodel. This CE not only references the compared blocks (in their original model), but also stores a similarity value for the comparison. We use a metric-based approach for computing the similarity, taking different properties, such as name, interfaces, or connected blocks into account. In Table 4.1, we show the relevant properties and how we compute the respective metric value. Moreover, we define a *weight factor* which is changeable and allows to define how much a certain property contributes ti the overall similarity value. With our family mining technique, we compare all model elements of one stage in the base models with all model elements of the corresponding stage of the comparison model and store the result in CEs.

Table 4.1: Properties of MATLAB/Simulink models and corresponding metrics used to compute the similarity (taken from [59]).

Property	**Weight**	**Computation**
name	5%	LD^* [34] of the blocks' names
function	75%	$sim(f_A, f_B) = \begin{pmatrix} 1 & type(A)=type(B) \\ 0 & else \end{pmatrix}$
#inports	5%	$\sum_{i \in IN} (i) / \lvert IN \rvert$
#inport-functions	5%	$\sum_{t \in T_{IN}} \left(\frac{\#t}{max(t)} \right) / \lvert T_{IN} \rvert$
#outports	5%	$\sum_{o \in OUT} (o) / \lvert OUT \rvert$
#outport-functions	5%	$\sum_{t \in T_{OUT}} \left(\frac{\#t}{max(t)} \right) / \lvert T_{OUT} \rvert$

*Levenstein Distance, IN/OUT - set of in-/outports of a model block

T_{IN}/T_{OUT} – set of functions of predecessor (IN)/successor(OUT)

This works reliably if corresponding elements are in the same stage and hierarchy. However, we observed that this is not always the case, especially, if subsystems are introduced in one of the models (thus, shifting elements to another hierarchy level, called *vertical dispersion)*) or if in one model, additional elements are added (*horizontal dispersion*). To overcome this limitation, we propose the MWT, which provides more flexibility for comparison across stages. In a nutshell, with MWT we loosen the definition of stages. Instead, we define a window with flexible size that we slide over the blocks of each model. Now, instead for each stage, we apply DFA for each window, that is, all blocks that are currently encompassed by the window. If comparisons are done, we move the window further, until we have considered each block in at least one comparison. For details about the algorithm of creating window pairs (i. e., one window for each model) and how to use them for comparison, see Schlie et al. [59].

After we are done with the comparisons, the CE elements are moved to the *matching phase*. In this phase, for each model element, we aim at identifying the best match, that is, from all CEs containing a particular model element, we search for the one with the *highest similarity value*. This is necessary, as the previous stage results into multiple CEs for each model element, and thus, is ambiguous with respect to identifying related model elements.

Finally, the best matching CEs are delegated into the *merging phase*. Here, the idea is to unify the compared models with explicit information about variation points. Currently, we devise a simple, metric-based decision process to decide on the variation point of two blocks. In particular, if the similarity value $sim \geq 0.95$, we assume the blocks to be *mandatory*. For $0 < sim < 0.95$, we declare blocks to be *alternative*, that is, mutually exclusive. In case that there is no similarity at all this indicates an *optional* block, i. e., this block exists only in one of the compared models. Since this way of merging also allows model elements in the merged model that exist only in one of the compared model, we refer to the resulting, merged model as *150%* model.

As long as there are more input models, we use this *150%* model as base model for the next iteration of the process with another comparison model. Otherwise, if all models have been compared, we export the final model, also providing a representation as *family model*, which allows for easier understanding of explicit variation points.

Quantitative Evaluation. For evaluating our technique, we use a mixed method approach. First, we present an empirical study where we apply our technique to real-world models and measure performance as well as accuracy (*quantitative evaluation*). Second, in the next part, we present the results of an interview study among professional model engineers, mainly aiming at insights about benefits and needs of variability mining in practice (*qualitative evaluation*).

For our quantitative evaluation, we apply both proposed techniques, that is, DFA only and DFA combined with MWT (for simplicity, only referred to as MWT in the remainder. The overall goal is to investigate (RQ 1) whether performance is reasonable and different when scaling up the size of models and (RQ 2) what level of accuracy (by means of precision and recall) we can achieve.

As input, we had two set of models. First, we had access to a real-world model of a *driver assistance system (DAS)* that has been made publicly available as part of a BMBF project. Based on the project documentation and domain knowledge of experts, we managed to decompose the DAS model into five sub models, together with particular dependencies that have to be considered when compose these sub models. We show the result of this decomposition in Table 4.2. We used these sub models to compose 19 large-scale models of different size, encompassing different combinations of the sub models and used these composed models within our evaluation.

As second set of models, we where provided with four models from an industrial partner: A pair of models of an *Exterior Light Front (ELF)* with 30 000 blocks each. And a pair of models of a *Drive Powertrain (DTM)* with 40 000 blocks each. With the help

Table 4.2: Basic properties of the extracted sub models from the SPES-XT study (taken from [59]).

Model name & *Abbreviation*	#blocks	$\#B_{Sub}$	$D_{Hierarchy}$
EmergencyBreak *'EB'*	409	43	7
FollowToStop (*req.* CC) *'FTS'*	699	77	11
SpeedLimiter *'SL'*	497	57	10
CruiseControl *'CC'*	671	74	11
Distronic (*.req* CC) *'DT'*	728	78	11

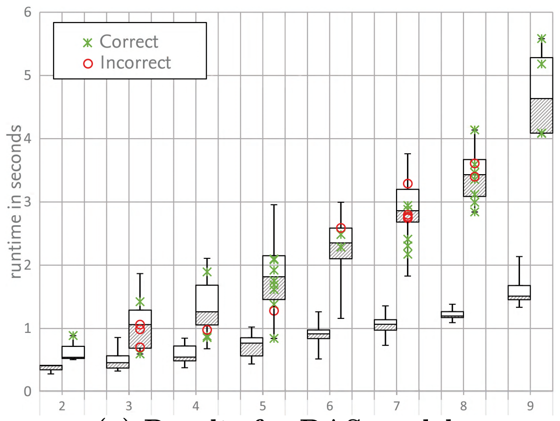
B_{Sub} – subsystem blocks, $D_{Hierarchy}$ – max. hierarchical depth, *req.* - requires

of domain experts, we could decompose these models in smaller sub models so that it could be processed by our family mining workflow.

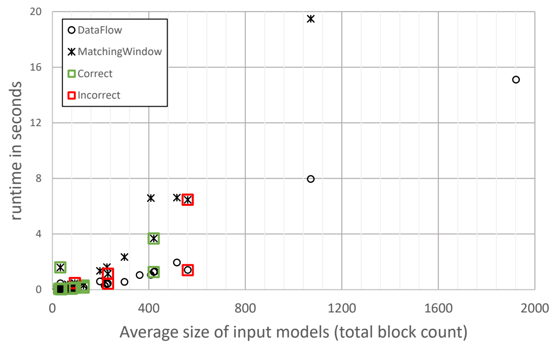
Given the set of models above, we conducted our study as follows, always using a pair of models as input for our family mining technique(s). For performance (by means of runtime) we applied our proposed process 10 times to take account for possible runtime deviations or warm-up effects. Moreover, we measure the overall runtime as well as how much time is spent in the particular phases. For accuracy, we face the problem that *no* ground truth is available for any of the models we used. Hence, we decided to provide a manual oracle as ground truth for a sample of all models/sub models. In particular, two experts analyzed 25% of the DAS models *before* we applied our family mining technique. Moreover, an expert from our industrial partner analyzed sub models or the ELF and DTM model, respectively. For all manually analyses, the focus was whether the assigned relation between blocks was understandable and according to the perception of variability for the analyst.

We show a summary of our results in Figure 4.3. Note that, for the results of the DAS models, we show pairs of box plots for each compared set of models with the left box plot representing results for the plain DAF approach and the right box plot representing the MWT approach. Moreover, for these models results are ordered from smallest to larges models (left to right). For performance, our results reveal two main results. First, both approaches have a quadratic increase of runtime when the size of models increase. Second, compared to the DAF approach, the MWT approach requires 148% more time on average for all models. However, even though being much slower, the MWT approach takes only seconds for the whole process, even for the largest models. This is also confirmed for the industrial models. Hence, we conclude that our technique scales well, and thus, is applicable to real-world models.

In Figure 4.3, we also illustrate the results of our manual inspection. Overall, approximately 70% of our manually analyzed models have also been correctly analyzed by our MWT technique. Moreover, correctly as well as incorrectly analyzed models are scattered over models of different size, and thus, do not reveal that the accuracy depends on the size of the models. A more detailed inspection of these results revealed that the incorrectly compared models originate in the merge phase, where blocks have not



(a) Results for DAS models



(b) Result for industrial models

Figure 4.3: Results regarding performance and accuracy for (a) the DAS models and (b) the industrial models (adopted from [59]).

been stored as expected. We conclude that the accuracy of the variability in compared models is reasonable, especially as incorrect results are mainly due to incorrect merge operations, but the other phases work as expected.

Qualitative Evaluation. To get more insights into how models are created and evolved, but also to qualitatively analyzed our findings above, we performed a survey and semi-structured interviews with eight domain experts in the area of model-based development. In particular, we were aiming at more detailed insights of use cases for variability mining, requirements of experts for such a technique, and also how far our technique fulfills these requirements.

The results can be divided into three categories. First, the *general expectations* of all participants are that our approach would support them in model maintenance and evolution. In particular, a lack of documentation about how models are related was mentioned as a specific use case, where our approach would be valuable. Moreover, participants acknowledged that a certain manual inspection, complementing our approach, is usually expected, and thus, minor issues with accuracy can be accepted. Finally, the capability of our approach to identify common but also variable parts was highlighted as beneficial, as simple diff or clone detection tools do not provide this information.

Second, different *fields of applications* have been confirmed by participants. Among others, using our technique for reengineering activities or to identify reusable artifacts has been highlighted in this category. Moreover, obtaining an initial family model that can be fed into a variant management tool (such as *pure::variants*) was mentioned as another application scenario. Among all participants, there was a consensus that especially in future development, when even more models are created, our technique exhibit huge potential.

Third, all participants confirmed the *usefulness of the results*, but were rather pessimistic regarding the scalablity of our approach (e. g., for a set of dozen models that are even larger than those currently analyzed by us). Some participants even appreciated the annotation of variability in the result model (especially for maintenance activities), whereas others would prefer a more high-level view.

In summary, the qualitative evaluation revealed that our approach meets main requirements and challenges of industrial practice and can support practitioners in evolving a set of similar models.

Generalization. While our family mining approach has received positive feedback even by practitioners, it has one central limitation: it is specifically tailored to MATLAB/Simulink. However, when experimenting with real-world models from industry, we observed that usually different block-based modeling languages are used as part of a model-driven development process. For instance, in the automotive domain, it is quite common that on lower hierarchy levels, MATLAB/Simulink function blocks contain *state charts* that describe the actual function. Hence, to comprehensively support variability mining for the whole model-driven process, support for other block-based languages is crucial.

Fortunately, even though different in detail, block-based modeling languages are generally similar regarding they underlying concepts and processes. Consequently, the overall concept and workflow of our family mining technique should also be applicable to other modeling languages than MATLAB/Simulink. We analyzed properties that are crucial to each language, and thus, must be adapted for different languages to be applicable within the family mining approach. As a result, we proposed a *generalization* that allows to customize our original variability mining technique for MATLAB/Simulink to any block-based modeling language. Next, I will briefly explain the five conceptual steps that are required to tailor variability mining to a particular modeling language. An overview of these steps, which also has been implemented by means of a customizable framework as part of the original work, is shown in Figure 4.4.

Step 1 – Analyze Language. A pivotal element of our family mining approach is the meta model for the respective language, as all steps (i. e., compare, match, and merge) are defined on concrete instances of this meta model. Hence, for a new block-based language to be used with family mining, we initially need a meta model, which can be obtained in two ways. First, we should search for an existing meta model for the subject language. In case that we find a suitable meta model, this not only saves time for the (manual) analysis of the language, but also makes the second step (building a meta model) superfluous.

If there is no meta model at hand, the language must be analyzed regarding its elements and all of its properties. This may be a tedious (manual) task that also requires some domain knowledge and language engineering capabilities. After this analysis, *relevant elements* that contribute to a language's functionality must be selected for the final

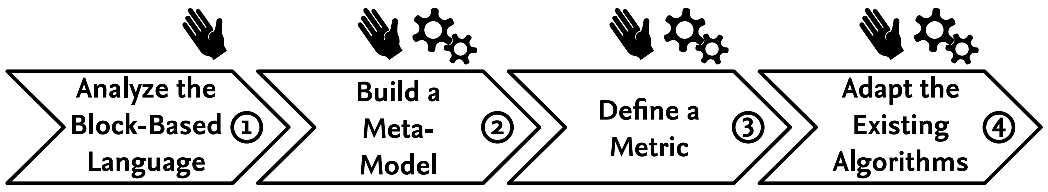

Analyze the Block-Based Language ①
Build a Meta-Model ②
Define a Metric ③
Adapt the Existing Algorithms ④

Figure 4.4: Conceptual steps for tailoring variability mining to an arbitrary block-based modeling language (adopted from [67]).

meta model. Put it differently, we consider elements being *irrelevant*, if they constitute only "syntactic sugar", and thus, can be transformed in an equivalent representation by means of other language elements. For each selected element, all of its properties that define the elements functionality have to be selected as well As an example, consider MATLAB/Simulink blocks: Here, properties such as *name, interface, and function type* define the functionality, and thus, are relevant, whereas properties such as *color or position* can be neglected. Once all relevant elements and its properties have been selected, we can proceed with the next step.

Step 2 – Build a Meta Model. The selected elements must now be composed to a tailored meta model that later is the starting point for the tailored family mining. Hence, besides choosing a suitable (meta) modeling language, developers have to choose the right relations between elements (modeled as *classes*) and add properties accordingly (as *attributes*). Moreover, a way must be defined to add variability annotations. Otherwise, it won't be possible to create the *150%* model that results from the merge phase in each iteration. In any case, if a well-formed meta model has been build, it should be able to express *semantically correct* models of the corresponding language. Finally, to make family mining work for the new meta model, *importer* and *exporter* are required that allow to parse input models but also to export them to an alternative/the original representation at the end.

Step 3 – Define a Metric. After building a proper meta model, the next crucial step is to define a suitable *similarity metric* that can be employed in the *compare phase* of the family mining approach. To this end, it is of superior importance to identify to what extent the different model elements contribute to the functionality of a model. Moreover, for each element, weights must be assigned to the properties in order to specify how important each property is for the *uniqueness* of this element. In this context, it is also important to choose a suitable value range for ranking these properties, as we have to normalize the computed similarities for an objective comparison between different values. In Figure 4.5, we show an example for such a ranking by means of a MATLAB/Simulink block element. As we can see, different properties are taken account for the comparison for such blocks, with the bock function being most important (weight value 0.75). Moreover, while the value range is between 0 and 1, we can see that all weights sum up to 1, and thus, allows for a normalized similarity values of blocks. Note that finding suitable weights for certain properties, especially when considered in

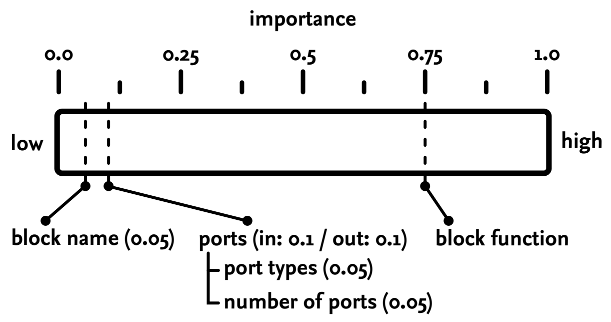


Figure 4.5: Exemplary ranking for properties of a MATLAB/Simulink block (taken from [67]).

concert, is not trivial. However, we argue that the necessary knowledge can be obtained during the first step (analyzing the modeling language).

Step 4 – Adapting the Algorithms. As a final step, the algorithms used in the particular phases of our family mining approach must be adapted to the newly introduced modeling language. For the *compare phase*, this affects the following algorithms. First, it must be defined which of the input models serves as a base model. Without further knowledge, selecting the smallest or largest model has been shown to be a suitable choice, which can also be set as default for new languages. Second, the traversal algorithm has to be adapted. In particular, elements that are suitable as *starting point* for the comparison have to be specified. This is also important for the subsequent definition of *stages*, since they must be aligned between the two compared models. Hence, if the initial model element is not selected properly, this may lead to the situation that stages are compared that do not correspond to each other. Third, we have to adapt the algorithm for dealing with hierarchies in models, as they may be realized differently for each language. Basically, we have to deal with three scenarios, given two compared models: No element exhibits a hierarchy, both element exhibit a hierarchy, and only one element of the compared models exhibits a hierarchy. Each of these scenarios must be properly handled by a particular algorithm and a suitable compare element must be defined as part of the meta model.

For the *matching phase*, we rely on the created compare elements of the previous phase, and thus, can abstract from concrete model elements. While this allows to use the same algorithm for all compare elements for a given language, this also means that the design of this element (and its corresponding class in the meta model) is crucial for the tailoring process. In particular, any logic that is related to concrete model element should be avoided. Apart from that, the algorithm for *handling of conflicts* (i. e., related compare elements with same similarity) must be considered for this phase, including an option for manual intervention by a developer or model engineer in case that the conflict can not be automatically resolved.

Finally, for the *merging phase*, the merge algorithm is language-specific, as it highly depends on the concrete model elements and their relations. Most notable, for merging hierarchies, the corresponding first level elements play a pivotal role. For instance, while blocks constitute such first-level elements in MATLAB/SIMULINK, regions constitute such elements in state charts, and thus, the algorithm has to be adapted according to the respective syntax and semantics. Moreover, the variation point (i. e., mandatory, optional, or alternative) is important for the merging algorithm, as it decides which elements have to be merged. For instance, in case that elements are identified as mandatory, only one element will be merged in the 150% model. Furthermore, the similarity thresholds for the particular variation points may be different for certain languages, and thus, have to be adjusted.

In our original work, we implemented these conceptual steps in a model-based fashion and used the resulting framework to successfully adapt our family mining approach to state charts but also to create an alternative technique for MATLAB/SIMULINK.

4.2 N-Way Comparison of MATLAB/Simulink Models with Static Connectivity Matrix Analysis

Although our family mining has been shown to be beneficial, we have seen two main limitations that may come into effect in certain situations. First, while still acceptable, our accuracy was at 70%, and thus, leaves room for an *increase in effectiveness*. Second, we have shown that the runtime of family mining increases with quadratic complexity. While this is no problem for a pair of models with reasonable size, it may become a major issue when comparing dozens of models at even larger scale. In particular, the fact that only two models are compared at the same time (thus, leading to multiple iterations) has been identified as bottleneck. For an increased efficiency, an *n-way algorithm*, comparing all models at the same time, is desirable. To this end, we propose SCMA, a technique for identifying similar structures (e. g., sub models) across multiple MATLAB/SIMULINK models while being more efficient and effective than the original family mining approach.

Preliminary Considerations. Our proposed technique relies on two fundamental insights. First, that a MATLAB/SIMULINK model and its hierarchies can be represented as a tree-like structure. In such a structure, each sub model constitutes a node at level δ_j and as a same time is the root node for its respective subtree. As a result, we can define the structure of a whole model in a recursive manner as a tree (cf. Figure 4.6).

Second, to allow for an efficient analysis of all models, we need a compressed representation that abstracts away most of the complexity of MATLAB/SIMULINK models while still containing the most important information. In other fields, *descriptors* have been used for such purposes that describe the most salient information but in a much simpler representation. In particular, *matrices* have been used as a common descriptor, as they allow to represent graph structures and exhibit numerical efficiency, and thus, enable large-scale analyses. As indicated in Figure 4.6, we can transform a MATLAB/SIMULINK model into a graph structure (here: a tree), which makes matrices also applicable as descriptor to our domain.

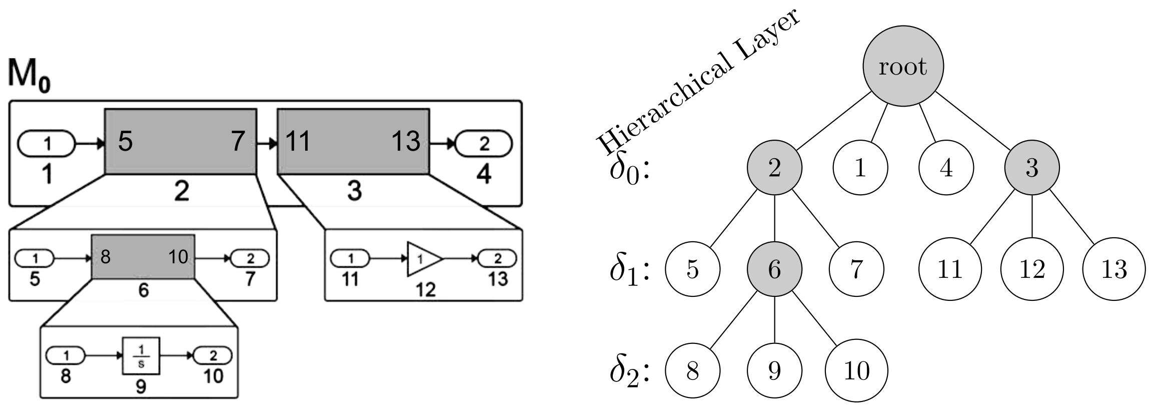
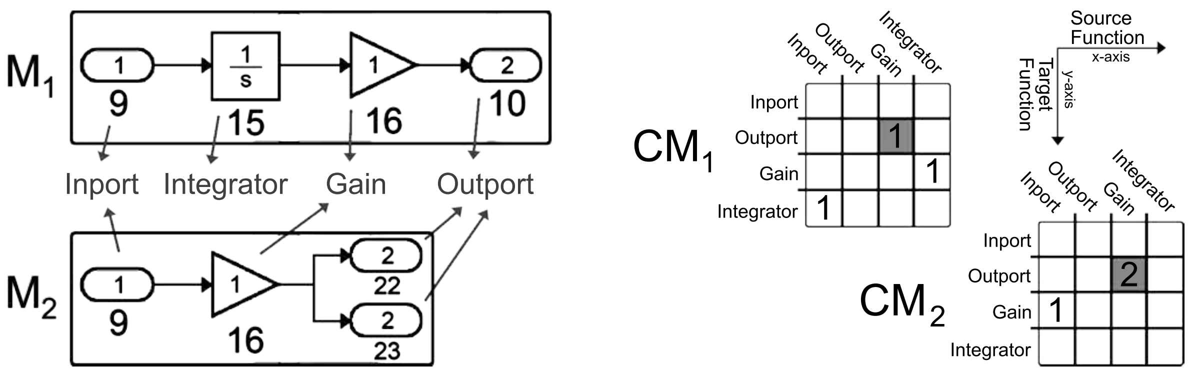


Figure 4.6: An exemplary MATLAB/Simulink Model and its graph representation (taken from [58]).

Static Connectivity Matrix Analysis. We show the overall workflow of SCMA in Figure 4.8 and briefly explain the particular steps next. As a first step, we have to transform the models to be compared in our descriptor, which is a *connectivity matrix (CM)*. With a CM, we employ the fact that each (sub) model in MATLAB/Simulink is a composition of directly connected blocks with each block having a specific function. More precisely, for any two sub models, we store in our CM which function blocks are connected with each other and how often. As an example, we show two MATLAB/Simulink models in fig:cmBeispiel (a) and their corresponding CMs in fig:cmBeispiel (b). Both models encompass four blocks, and thus, both matrices are sized $4x4$, as each block needs to be present at the x-axis (constituting the *source function*) and y-axis (constituting the y-axis).



(a) MATLAB/Simulink Models M_1 & M_2 **(b) Corresponding CMs for M_1 & M_2**

Figure 4.7: Two exemplary MATLAB/Simulink models and their corresponding CMs (taken from [58].

In the cells of each matrix, we denote existing *connections* between blocks, that is, edges in the model that represent a signal going from the source function to the target

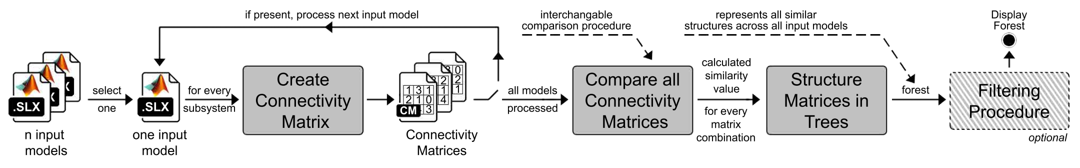

if present, process next input model
interchangable comparison procedure
represents all similar structures across all input models
Display Forest
.SLX
n input models
select one
one input model
for every subsystem
Create Connectivity Matrix
CM
Connectivity Matrices
all models processed
Compare all Connectivity Matrices
calculated similarity value
for every matrix combination
Structure Matrices in Trees
forest
Filtering Procedure
optional

Figure 4.8: Overall workflow of our static connectivity matrix analysis (taken from [58]).

function. For instance, in model M_1, a connection exists from the block *Gain* to the block *Outport*. Consequently, the corresponding CM_1 has an entry in the cell that represents this connection. Moreover, if more than one connection exists between two blocks, we simply increase the value for the respective cell, as done in our example for CM_2.

To ensure that all CMs have the same dimension, we create a dictionary of all distinct blocks over all models during matrix instantiation. As a result, all connectivity matrices are created with dimensions according to the length of the dictionary. For instance, both CMs in Figure 4.7 contain the block *Integrator*, although this block is only present in model M_1. Along with the process of creating the dictionary, we also retrieve and store all connections between any two blocks of the models. Moreover, we maintain hierarchical dependencies during this process. In particular, if a certain sub model contains further sub models, than the resulting CMs are also stored in this hierarchical order by means of a tree structure. This allows us in later phases, two compare sub models even across hierarchical levels.

For *comparing all modells*, we now solely rely on the matrix representation and do not have to take the original models into account anymore. More precisely, we compare any CM from model M_x with all other CMs of the other models, except those from M_x. For each CM comparison, we compute a similarity value $0 \leq \omega \leq 1$. The concrete algorithm for this comparison and similarity computation can be found in [58]. For a brief illustration, let's ahve a look at our example in Figure 4.7. Comparing the entries for the two blocks *Gain* and *Outport* would yield a similarity value of 0.5. For both matrices CM_1 and CM_2, we yield a similarity value $\omega = 0.125$. The similarity values of all comparisons are stored in a table, where each cell contains the similarity value for a certain pair of CMs. In case that more than two models are compared, already this table allows to identify parts that are identical or similar across all of these models. However, in this representation, the information about hierarchies is lost.

To preserve this information, be propose *forest creation* (cf. Figure 4.8) as a further step, that is, structuring the matrix comparisons in trees. The main idea here is that we group similar CMs in nodes based on a user-defined threshold $omega_min$. Moreover, if CMs of a certain node exhibit a parent-child relationship with CMs of another node, we connect these nodes, and thus, successively create a tree structure that resembles the hierarchical structure of the original models. Since the CMs of a particular model may exhibit a different similarity with CMs of the other models, this necessarily leads to the fact that a pair of CMs, depending on their similarity value ω are either inserted into an existing node, used to create a new node that is inserted in an existing tree, or used to create an entirely new tree. Hence, at the end of this step, we end up with multiple trees, eventually forming a forest.

This forest is now subject to the last step, that is, filtering out nodes (or entries therein) depending on certain criteria. For instance, a common scenario is to filter out irrelevant CM comparisons, that is, those who fall below a user-defined threshold. This, in turn, allows to focus on rather similar or even identical CMs, and thus, identify commonalities in the original models. Another possibility is to filter out nodes that contain CMs of models that are currently not of interest. This filtering method allows to focus on a subset of all compared models and the commonalities and differences between these models. In general, filtering procedures can be considered as *projection* and may even lead to whole trees disappearing.

Evaluation. For evaluating our technique, we used the DAS models, which we already used and explained in Section 4.1. The difference is that we no only compared pair of models, but also larger sets up to 19 models at once. With this empirical setup, we address three aspects of superior interest for demonstrating the applicability of our technique: RQ 1 Suitability of our chosen descriptor, RQ 2 precision and recall of SCMA, and RQ 3 performance in terms of runtime and scalability.

For the first RQ, we evaluate the algorithmic complexity for creating our CMs as well as the correctness of the created CMs (i. e., whether they adhere to their counterpart in the original model). For RQ 2, we manually investigated 18 comparison (i. e., results after applying SCMA with combinations and amount of input models), in particular, the trees that have been created. Finally, for RQ 3, we take algorithmic complexity as well as the runtime of SCMA and its mandatory phases into account.

For creating all CMs, the need for parsing all models and creating the dictionary is clearly the most expensive procedure. Hence, we measured the time needed for this step for all set of models in our evaluation. Our results reveal that eve for the largest set of models, comprising 19 model variants, the time for creating a total of 1 528 CMs lies in the order of milliseconds. Hence, we argue that our descriptor is easy and efficient to extract from MATLAB/SIMULINK models. For correctness, we focussed on possible mismatches between original sub models and corresponding CMs. To this end, we reviewed the distribution of similarity values when comparing all 19 model variants (cf. Figure 4.9). Our data reveal that essentially two peaks show up: Around 80% of the comparisons do not yield any similarity ($\omega = 0$) wheres 5% of the comparisons indicate that the respective CMs are identical ($\omega = 1$). We selected 2 000 comparisons for each of these peaks for manual inspection with the result that 100% of comparisons for $\omega = 0$ where correctly and 93% of the comparisons for $\omega = 1$ being correctly (for the remaining 7%, corresponding SMs where not identical). The remaining 15% distribute evenly in the range $0 < \omega < 1$ and we manually inspected 400 comparisons that confirmed that our comparison based on CMs is correct.

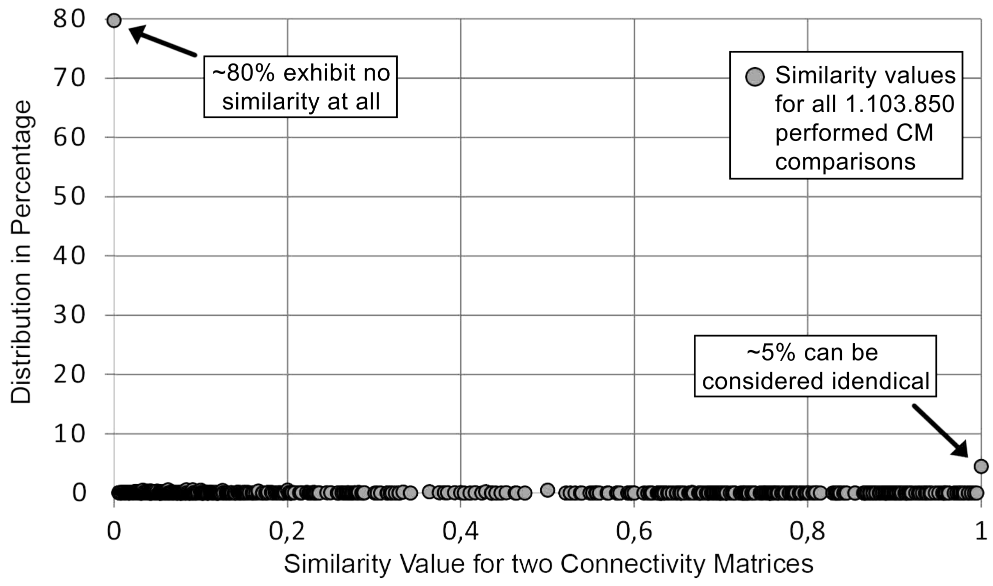


Figure 4.9: Distribution of similarity values for $\approx$ 1.1 million CM comparisons (taken from [58]).

For RQ 2, we manually inspected the generated trees for 18 model comparisons. Since the amount of trees can explode, even for a small amount of models, we set the similarity threshold for filtering during forest creation to $\omega = 1$, that is, we only inspected those trees that contain nodes of identical CMs. The resulting trees manually analyzed ranged from size one to size thirty (i. e., comprising thirty nodes). For both, precision (i. e., nodes have been identical) and recall (i. e., no CM comparison is missing), our inspection revealed that all trees have been correctly created. Hence, we argue that our technique is precise and results in a high recall.

Regarding RQ 3, we first evaluated the algorithmic complexity of the three mandatory phases of our technique, that is, CM creation, CM comparison, and forest creation. While for the first phase exhibits a linear complexity (wrt. number of blocks), the two other phases come with quadratic complexity regarding the number of CMs. Hence, our overall technique has a quadratic complexity. For the runtime, we illustrate in Figure 4.10 how this changes depending on the number of CMs, which is influenced by different similarity thresholds ω_{min}. As our results reveal, for the largest set of models (i. e., 19 model variants) and a threshold of $\omega_{min} = 0$ (i. e., all CM comparisons are considered) we achieve an overall runtime of 47 minutes. However, if we focus on more similar CMs, which are generally of greater interest, by setting $\omega_{min} = 0.4$, we can reduce the runtime significantly to less than 5 minutes. For even higher thresholds, the runtime decreases further. Having a closer look, how this runtime distributes over the three phases, our results reveal the following. Approximately 3.4% of the time is used for CM creation, whereas CM comparison accounts for 0.8% of the overall runtime. Hence, the by an order of magnitude largest effort is spent in the phase of forest creation,

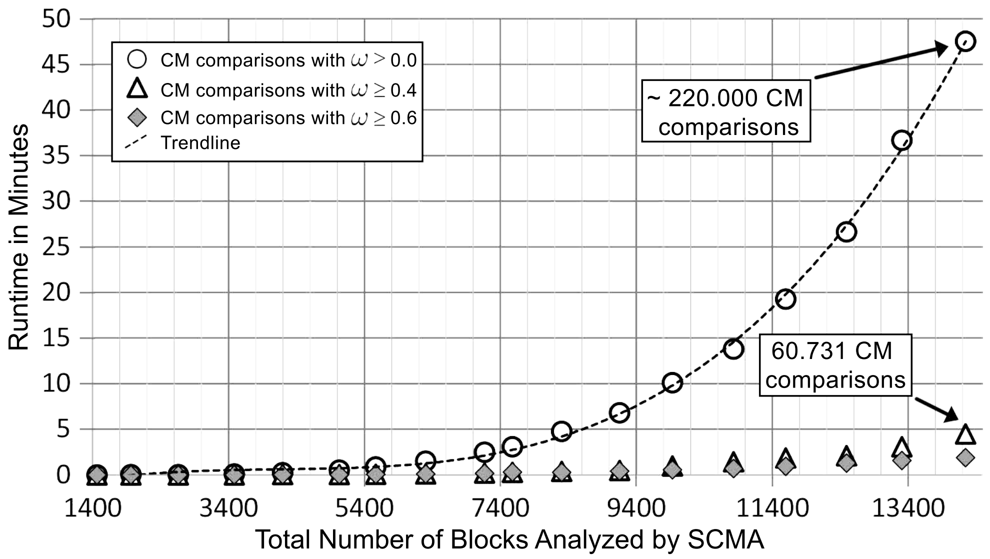


Figure 4.10: Runtime of SCMA relative to the number of CMs (taken from [58]).

accounting for 95.8%. However, since this is also the phase where filtering takes place, the similarity threshold can drastically decrease this time, which makes our technique scalable to some degree.

Conclusion. In conclusion, we argue that our technique is able to compare multiple MATLAB/Simulink models at once, even at large scale. Moreover, our results show that our technique is precise and comes with a competitive runtime, mainly due to our chosen descriptor.

4.3 Reverse Engineering Variability from Natural Language Requirements

Requirements or *Software Requirements Specification (SRS)* are usually the very first artifact that is created when starting a software project. It results from intensive discussions with customers about all facets of the software system to be developed, mostly in form of plain text. Hence, if properly done, requirement specifications reflect on all features of a software system as well as how these features depend on each other. Consequently, requirements are a valuable source for extracting variability to support systematic reuse in variant-rich software systems.

In this Section, we propose two techniques for variability mining of SRS, one using probabilistic relevance and word embedding and another, preliminary technique using neural networks. Initially, I briefly report on an initial literature survey to evaluate the current state-of-the-art and to identify gaps therein.

Preliminary Literature Review. Being aware that already certain techniques exist to extract features from NLR, we conducted a *Structured Literature Review (SLR)* to obtain an overview of the state-of-the-art and to identify gaps and weaknesses of currently available techniques [36]. In particular, we wanted to know RQ 1 what concrete techniques have been proposed, RQ 2 how these techniques are supported regarding tools and automation, and RQ 3 how reliable these techniques are for feature extraction.

To this end, we conducted an SLR according to the guidelines of Kitchenham [29]. We gained several insights, form which the following have been of superior interest regarding the subsequent development of our own techniques.

For RQ 1, we observed that a variety of techniques have been applied, and that most of them come from the field of *Natural Language Processing (NLP)*. Moreover, our results indicated that the combination of NLP techniques is likely to increase the quality of the results. Another important aspect we revealed is that for variability extraction, it is crucial to not only analyze NLR regarding their syntax, but also to understand them semantically. Finally, our data revealed that combining the results of NLP with clustering algorithms is beneficial, as the latter allows to identify relationships between detected features such as groups or parent-child relations.

For RQ 2, we made two main observations. First, if tools have been provided, they make a reliable and mature impression. However, for the vast majority of reviewed approaches, no tool has been provided at all, and thus, mitigating replicability and reproduceability for the proposed techniques. Second, most of the proposed techniques come with a high degree of automation, although most of them are not fully automated. While we agree that some manual intervention is necessary, we nevertheless see some room for increasing the degree of automation for certain tasks.

Finally, for RQ 3 the following observations have been made. First, the accuracy it the most critical problem of all approaches available so far. While this still allows for supporting developers in their manual inspection, the remaining effort is considerable high due wrong results of the existing techniques. Second, most techniques are only able to extract partial variability information that not even allows to recreate a feature model. Most notably, while features could be detected, their relationship is mostly missing. Finally, in most cases a comprehensive and reliable evaluation is missing, thus, raising doubts in the trustworthiness of the proposed techniques.

Given these insights, we concluded that especially regarding the accuracy , the tool support, and the evaluation, there is a huge gap in the applicability and reliability of current approaches and started to develop a technique on our own.

Automated Feature Extraction with Neural Networks. For our first technique, we employ NLP techniques as well as deep learning techniques, in particular neural networks. The rationale was that, while NLP can provide us with several algorithms to focus on the most important information and extract important semantics, a neural network can be trained in order to identify recurrent patterns which then could be used to identify features and establish relations among them. An overview of the technique is shown in Figure 4.11 (a) and briefly described in the following.

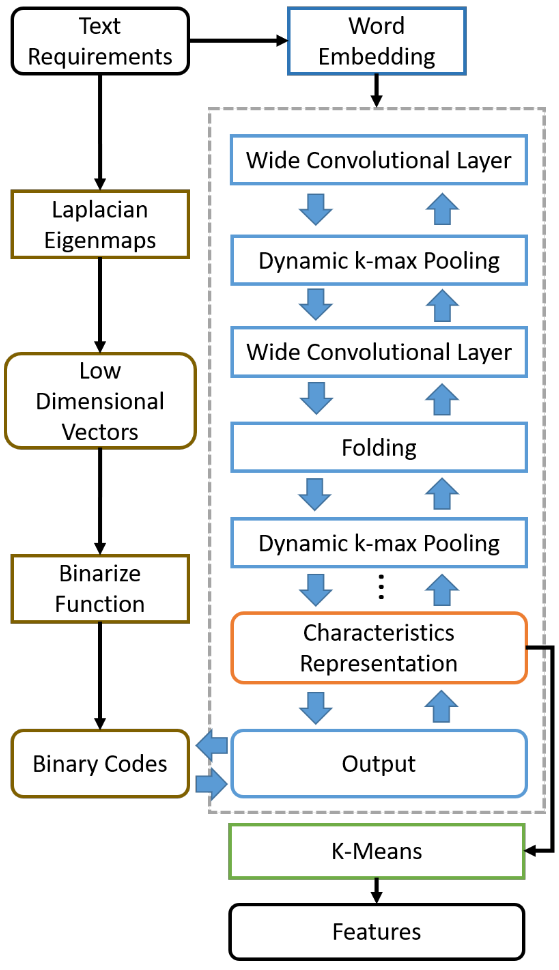
Initially, we preprocess our requirements to decompose them in single sentences, but also to remove unimportant or even misleading information. Among other, common NLP techniques such as stemming or stop word removal are applied in these steps. Afterwards, the resulting requirements are processed in two ways. First, we apply *Laplacian Eigenmaps*, an algorithm for dimensionality reduction, to the requirements to obtain a low dimensional representation. The reasons are twofold: First, we want to narrow down the dimensions, representing the properties of our requirements, to those that contain most of the information needed or the extraction process. Second, we can convert the resulting, low-dimensional matrix into binary codes, which then can be easily compared with the output of our neural network during the training phase.

In parallel to the dimensionality reduction we feed our preprocessed requirements into a *Dynamic Convolutional Neural Network (DCNN)*, which has been especially proposed for modeling sentences. To this end, we first apply a word embedding model to our requirements in order to obtain a vector representation for each word of the requirements. Subsequently, these word vectors are transformed into a matrix, which is then fed into the DCNN. Within the DCNN, the matrix is processed by several convolutional layers and at the end of this process, the result is compared with the binary codes of the *Laplacian Eigenmaps* algorithm. This process is repeated several times to learn the precise characteristics of our requirements.

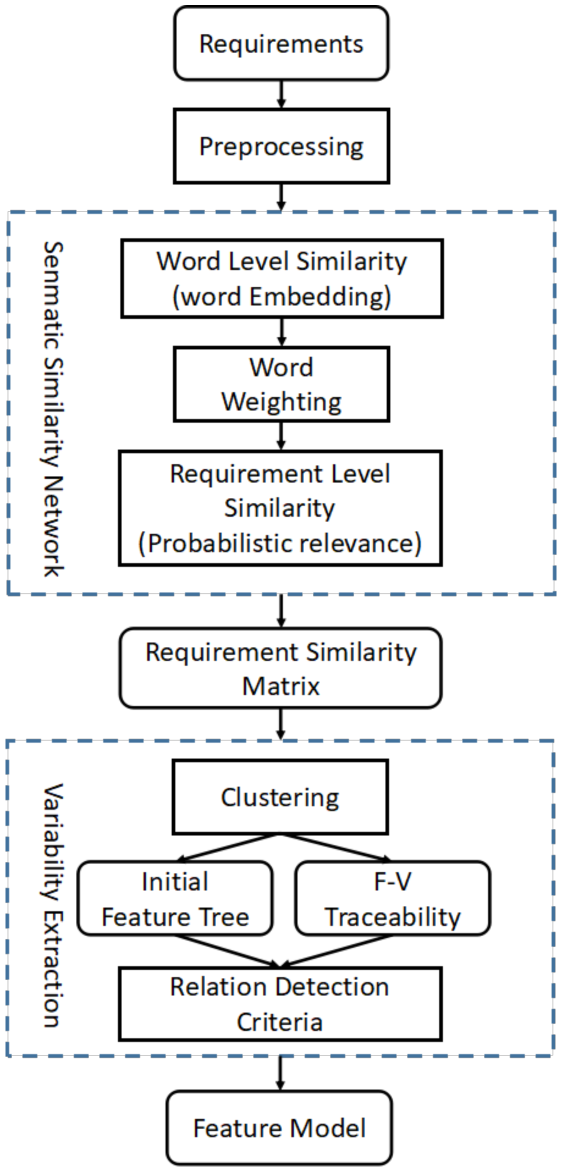
Once the DCNN has finished, we feed the result into a clustering algorithm (in this case k-nn). The idea here is to group sentences that exhibit similar characteristics with the intuition that one group can be interpreted as a feature. We conducted a preliminary evaluation for a given set of requirements from a *Body Comfort System (BCS)* that have been made publicly available. The BCS comprises 95 requirement with 117 sentences and also comes with a feature model that serves as a ground truth. However, due to the very small amount of requirements, we observed a high loss in accuracy which resulted in only few features correctly detected. The main reason we identified for this loss is the fact, that our technique requires a large amount of data to learn about the characteristics of requirements, which is not possible with only about hundred requirements.

Extracting Variability Based on Probabilistic Relevance and Word Embedding. As our first technique is not applicable to the requirements available for us, we developed an alternative technique, which is shown in Figure 4.11 (b). Given a set of requirements from different variants, which are preprocessed as mentioned above, this technique consists of two parts: A *semantic similarity network* and a part for *feature & variability extraction.*

For the first part, we apply different techniques to determine the semantic similarity of requirements. Initially, we apply Word2Vec, a word embedding model to obtain



(a) **Workflow of our proposed deep learning technique**


Requirements
Preprocessing
Senmatic Similarity Network
Word Level Similarity (word Embedding)
Word Weighting
Requirement Level Similarity (Probabilistic relevance)
Requirement Similarity Matrix
Variability Extraction
Clustering
Initial Feature Tree
F-V Traceability
Relation Detection Criteria
Feature Model

(b) **Workflow of our probabilistic relevance technique**

Figure 4.11: Respective workflow of our two proposed techniques for variability extraction from NLR (taken from [35, 37]).

a vector representation of each word. In particular, this model takes the semantic similarity of words into account, and thus, word vectors of similar words are located close to each other in the resulting vector space. Within this vector space, we compute *cosine similarity* between two word vectors to determine the similarity of the corresponding words.

Afterwards, we extend this *word level similarity* by two further characteristics, that is, *distribution* of words in each requirement and *significance* of each word. To this end, we make use of a probabilistic relevant framework called BM25+. In a nutshell, BM25+ performs *term frequency – inverse document frequency (TF-IDF)*, a weighting technique that measures how much a word contributes to the relevance of two texts. Since even BM25+ exhibits certain limitations, we extend this step by topic words and the *inverse document frequency (IDF)* to compute weights for each word in the requirements. In particular, we assign a predefined value for the weight, if the word is in the set of topic words. Otherwise, we compute the IDF of the word and use the result as weight value for the word. This way, we can control how much a specific word contributes to the overall similarity of a pair of requirements Based on the probabilistic relevance and the additional weights for each word, we compute the final *requirement level similarity* (cf. [37] for details).

In the second part of our technique, we make use of the information about requirement similarity to extract features and variation points (i. e., dependencies between features). To extract features and create a tree-like feature diagram, we employ *Hierarchical Agglomerative Clustering (HAC)*, which groups similar requirements into the same cluster and constructs a hierarchy between these clusters, used by us to create the feature diagram. To this end, we make first use of the similarity between each pair of requirements, as explained above, as clustering criterion. Moreover, as a second aspect, we also consider the dissimiliarity of requirements, computed as pairwise distance, to find the closest pair of requirements, respectively, and merge them into a single group (i. e., set of requirements). In further steps, we (A) check for possible wrongly detected hierarchies and flatten them using a *inconsistency coefficient* and (B) define an inconsistency threshold to prevent from inaccurate clustering of requirements.

After we obtained the feature diagram, we make use of it to define criteria for extracting variation points. We show the process of this final step in Figure 4.12. As a pivotal element of this process, we also employ information from the very beginning of our process, that is, which requirement originates in which variant. Since requirements are now also mapped to features we can, eventually, also establish a mapping between the features and the original variants (the *F-V traceability*), with the requirements serving as kind of a proxy.

We propose the following four criteria for variation point detection:

1. We consider a feature to be *mandatory*, if it is (a) contained in *all variants* and (b) the same holds for all of its sub features (i. e., the subtree).

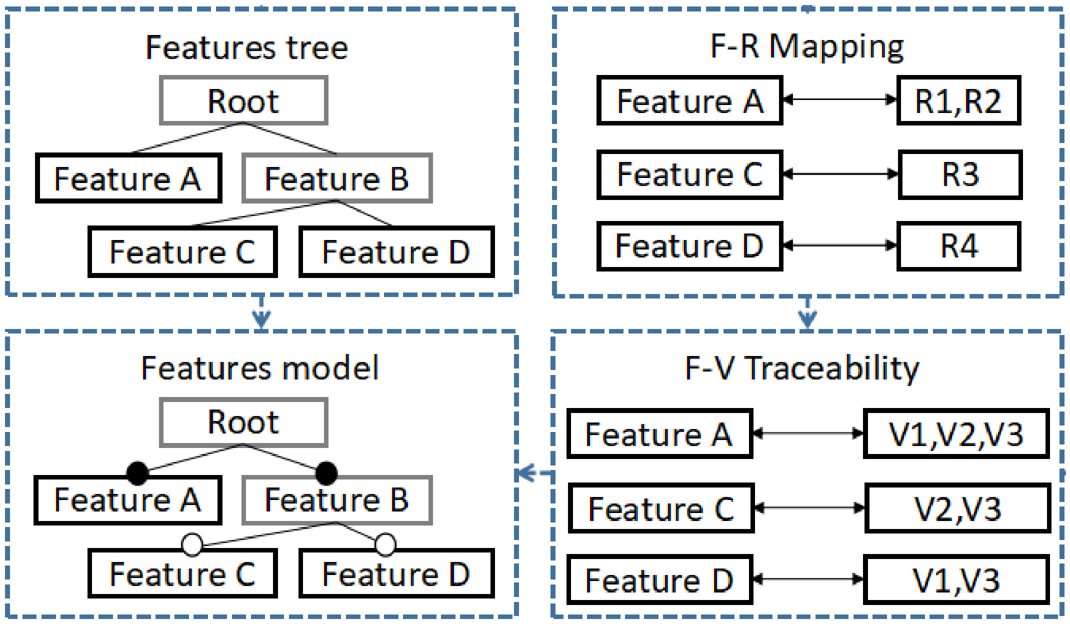


Figure 4.12: Overview of the process for identifying variation points (adopted from [37]).

2. If at least one of the aforementioned conditions is not fulfilled, we consider a feature as *optional*

3. For features forming an OR group, they must have the same parent feature and at least $(n-1)$ pairs of these sibling features must be contained in the same variant. Beyond that, we require that (a) all of these pairs of features are unique, (b) the $(n-1)$ pairs must from the transitive closure for all features under this (same) parent features, and (c) these n features have to cover all variants from which the input requirements have been taken.

4. Finally, to specify a group of features as XOR group, the union of these features must cover all variants *and* no two features occur together in a particular variant.

Consider the example in Figure 4.12. Since feature A is part of all input variants and has no sub features, it is clearly mandatory. Also, features C and D are optional, as they do not occur in all variants. For feature B, no information exists about corresponding variants, since it is used mainly for grouping the sub features C and D. However, since the union of these features also covers all input variants, also feature B is considered to be mandatory.

To evaluate the applicability of our approach, we conducted an empirical study with the requirements of the already mentioned BCS. In particular, we where interested in the accuracy of our technique by means of precision, recall, and f-measure. As BCS already has a feature model, we can (a) use this as ground truth and (b) create variants which we use as input. Hence, we selected 19 variants that have been used in previous studies by other researchers, and used the corresponding set of requirements from each variant (a subset of all requirements available) as input to our technique. Moreover, we used a pre-trained WORD2VEC model based on the Google News dataset, which comprises 100 billion words and set certain parameters for the used algrithms (cf.[37]).

We then applied the above mentioned process, resulting into a feature model with 23 concrete and four abstract features. Moreover, our variation point detection resulted into eight features being *mandatory* and 19 being *optional*, whereas neither OR nor XOR groups could be identified. We compared this resulting feature model with the ground truth (i. e., the original feature model of BCS) to evaluate the accuracy.

For precision (i. e., correctly identified features), we achieve 0.70, which measn that the majority of features has been identified correctly. For the false positives (i. e., features that are wrongly identified by our technique), we found out that the main reason is that we created to fine-grained features, and thus, created to many sibling features for two parent feature, *LED* and *Security*, respectively. We argue that these wrongly identified features still belong to a distinct part of the feature model, and thus, may be easily merged by manual intervention, which usually takes place in such a process.

For recall, we even achieve 0.79, as we only miss five feature at all. This is mostly the case, because some of our features are too coarse-grained, that is, we group requirements to one feature that is separated in two features in the ground truth. Again, this is only a minor threat, as the features in the ground truth are logically connected (e. g., as sibling feature), and thus, can be separated in our feature model manually. Setting precision and recall in relation, we achieve an F-measure of 0.74, which is very good as it indicates that both values, precision and recall, are relatively high.

Finally, for the variation points, we missed one OR and XOR group, respectively. Instead, we marked two features, actually belonging to one of these groups as mandatory, which results in two more mandatory features in our feature model compared to the ground truth.

Besides accuracy, we achieve a high degree of automation. Manual intervention is only necessary at the beginning, when parameters for algorithms such as BM25+ and HAC have to be set. Nevertheless, to keep control over the whole process and even improve the result, more intervention, such as integrating additional domain knowledge, is possible.

In summary, we argue that our technique overcomes the limitations that we identified for other techniques in our SLR. We provide a highly automated technique that automatically generates a feature model and identifies variation points based on heuristics. Moreover, even with a rather small dataset, we achieve very good results. Nevertheless, there is still a plenty of improvements possible. First, applying the technique to larger NLR from different domains is necessary to demonstrate generalizeability. Second, sophisticated NLP techniques can help to suggest more precise feature names. Third, improving the variation point detection, especially groups and cross-tree constraints is an important aspect that must be covered in future.

5. Conclusion

In this chapter, I briefly summarize the contributions of this thesis and provide a brief outlook on future work.

Generally, the contributions of this thesis are all concerned with the challenges of evolving variant-rich software systems. While both, structured as well as adhoc reuse, provide several benefits, they also come with disadvantages that impede different activities during the evolution of such systems.

For structured reuse, this is mainly caused by the intrusiveness of integrated variability and the related variability mechanisms. As these mechanisms, mainly the C preprocessor, interact with the host language, they hinder program comprehension, increase maintenance effort, and may be prone to errors.

To address these challenges, we made three contributions. First, we proposed the notion of variability-aware code smells to better describe high-level pattern of problematic usage of variability mechanisms, mainly for CPP and . In particular, we derived several of these variability smells, based on original code smells, described their pattern and possible disadvantages. We also conducted a survey, which confirms that these smells exist and can have a considerable impact on evolutionary activities. Second, we proposed a metric-based detection technique that enables the automatic detection of such smells. By means of the smell AnnotationBundle, we introduced metrics that affect this smell and also presented a parameterizable detection tool. We evaluated our detection technique with open-source systems, revealing that these smells can be found frequently in these systems, especially when considering the Top-10 results for the corresponding smell metric. Third, we investigated the impact of preprocessor usage on maintenance effort by means of statistical analysis. To this end, we used the particular metrics, introduced for the AnnotationBundle smell, and analyzed their correlation of two maintainability measures, frequency and amount of changes. Our results reveal only a small to medium effect of how and when the preprocessor is used, whereas the size of functions turned out to have a greater impact on maintainability.

For adhoc reuse, the challenges are mainly caused by redundant and concurrent development of variants and the missing information about commonalities and differences between these variants.

To mitigate these reasons, we made two contributions. First, we proposed a model-based technique, family mining, to identify commonalities and differences among MATLAB/Simulink models. As a result of this technique, we can relate model elements form different models with each other, also specifying their variability type. The evaluation of our technique confirmed its applicability even to real-world models and also revealed a reasonable accuracy of 70%. Moreover, we conducted a survey and interviews with modeling experts, stating that our technique has great potential especially for maintaining and evolving model variants to be created in future. Moreover, we proposed a concept how to customize this technique to other block-based modeling languages. As this technique also had some limitations, we proposed another technique that is able to compare multiple models at once, by using a connectivity matrix as an abstract representation of models. With an evaluation, we could show that this technique has a high precision and recall, and even for multiple models, provides scalability towards industry-sized models.

Second, we proposed two techniques for extracting variability from natural language requirements that rely on natural language processing and machine learning techniques. While the deep learning technique could not be positively evaluated (mainly due to missing data), our technique with probabilistic relevance has shown a high precision *and* recall (compared to existing approaches) for creating a feature model, and thus, is applicable at least to smaller sets of requirements. This could considerable improve the recreation of variability information from requirements and also, due to traceability links, map this information to other artifacts.

For future work, the idea is to analyze other quality aspects, such as error-proneness or vulnerabilities, in relation to integrated variability and also to develop analysis techniques that support efficient regression testing of such systems. For adhoc reuse, the focus is on combining information from variability mining of different artifacts, and thus, to obtain a more comprehensive picture of commonalities and differences. Moreover, techniques to feed this information back into the development process are subject to future research.

A. Appendix

In the following, I list selected publications that have been summarized within this thesis. These publications are listed in chronological order of appearance and a preprint of each paper (in the same order) is attached to this thesis. Moreover, for each publication, I summarize my individual contribution to the paper.
NOTE: order of authorship in all papers below is according to extent of contributions.

[65] S.Stănciulescu, S.Schulze, A.Wąsowski. Forked and Integrated Variants in an Open-Source Firmware Project". In: *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 151–160.
Contribution: Conceptualization (30%), survey/interview creation (20%), Writing: Introduction/Conclusion (30%), Concept Presentation (40%), Evaluation (30%), Related Work (15%); Reviewing and Supervision.

[17] W. Fenske and S. Schulze. "Code Smells Revisited: A Variability Perspective". In: *Proc. Int'l Work. on Variability Modeling of Software-Intensive Systems (VaMoS)*. ACM, 2015, pp. 3–10.
Contribution: Conceptualization (50%), Survey Creation (50%), Writing: Introduction/Conclusion (60%), Concept Presentation (40%), Evaluation (40%), Related Work (25%); Reviewing and Supervision.

[18] W. Fenske, S. Schulze, D. Meyer, and G. Saake. "When Code Smells Twice as Much: Metric-Based Detection of Variability-Aware Code Smells". In: *Proc. Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2015, pp. 171–180.
Contribution: Conceptualization (50%), Design & Implementation (20%), Writing: Introduction/Conclusion (40%), Concept Presentation(40%), Evaluation (40%), Related Work (25%); Reviewing and Supervision.

[19] W. Fenske, S. Schulze, and G. Saake. "How Preprocessor Annotations (Do Not) Affect Maintainability: A Case Study on Change-Proneness". In: *Proc. Int'l Conf. on Generative Programming: Concepts Experiences (GPCE)*. ACM, 2017, pp. 77–90.
Contribution: Conceptualization (50%), Design & Implementation (20%), Writing: Introduction/Conclusion (40%), Concept Presentation (40%), Evaluation (40%), Related Work (25%); Reviewing and Supervision.

[59] A. Schlie, D. Wille, S. Schulze, L. Cleophas, and I. Schaefer. "Detecting Variability in MATLAB/Simulink Models: An Industry-Inspired Technique and Its Evaluation". In: *Proceedings of the International Systems and Software Product Line Conference (SPLC)*. ACM, 2017, pp. 215–224.
Contribution: Conceptualization (30%), Writing: Introduction/Conclusion (60%), Concept Presentation (25%), Evaluation (40%), Related Work (20%); Reviewing and Supervision.

[67] D. Wille, S. Schulze, C. Seidl, and I. Schaefer. "Custom-Tailored Variability Mining for Block-Based Languages". In: *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2016, pp. 271–282.
Contribution: Conceptualization (30%), Writing: Introduction/Conclusion (60%), Concept Presentation (25%), Evaluation (30%), Related Work (10%); Reviewing and Supervision.

[58] A. Schlie, S. Schulze, and I. Schaefer. "Comparing Multiple MATLAB/Simulink Models Using Static Connectivity Matrix Analysis". In: *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 185–196.
Conceptualization (40%), Writing: Introduction/Conclusion (60%), Concept Presentation (40%), Evaluation (50%), Related Work (30%); Reviewing and Supervision.

[36] Y. Li, S. Schulze, and G. Saake. "Reverse Engineering Variability from Natural Language Documents: A Systematic Literature Review". In: *Proceedings of the International Systems and Software Product Line Conference (SPLC)*. ACM, 2017, pp. 133–142.
Contribution: Conceptualization (70%), Conducting SLR (50%), Writing: Introduction/Conclusion (60%), Concept Presentation (40%), Evaluation (40%), Related Work (25%); Reviewing and Supervision.

[35] Y. Li, S. Schulze, and G. Saake. "Extracting Features from Requirements: Achieving Accuracy and Automation with Neural Networks". In: *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 477–481.
Conceptualization (40%), Writing: Introduction/Conclusion (60%), Concept Presentation (40%), Evaluation (50%), Related Work (30%); Reviewing and Supervision.

[37] Y. Li, S. Schulze, and G. Saake. “Reverse Engineering Variability from Requirement Documents based on Probabilistic Relevance and Word Embedding”. In: *Proceedings of the International Systems and Software Product Line Conference (SPLC)*. ACM, 2018, pp. 121–131.
Conceptualization (40%), Supervision of Implementation, Writing: Introduction/-Conclusion (60%), Concept Presentation (40%), Evaluation (40%), Related Work (30%); Reviewing and Supervision.

[1] I. Abal, C. Brabrand, and A. Wasowski. “42 Variability Bugs in the Linux Kernel: A Qualitative Analysis”. In: *Proc. IEEE/ACM Int’l Conf. on Automated Software Engineering (ASE ’14).* ACM, 2014, pp. 421–432.

[2] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, Ş. Stănciulescu, A. Wąsowski, and I. Schaefer. “Flexible Product Line Engineering with a Virtual Platform”. In: *Companion to the Proc. Int’l Conf. on Software Engineering (ICSE ’14).* ACM, 2014, pp. 532–535.

[3] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines – Concepts and Implementation.* Berlin Heidelberg, Germany: Springer, 2013.

[4] S. Apel, C. Kästner, and C. Lengauer. “Language-Independent and Automated Software Composition: The FeatureHouse Experience”. In: *IEEE Trans. Softw. Eng.* 39.1 (2013), pp. 63–79.

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer. “Scaling Step-Wise Refinement”. In: *IEEE Trans. Softw. Eng.* 30.6 (2004), pp. 355–371.

[6] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wąsowski. “Three cases of feature-based variability modeling in industry”. In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS).* Springer, 2014, pp. 302–319.

[7] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wąsowski. “Feature-to-Code Mapping in Two Large Product Lines”. In: *Proceedings of the International Software Product Line Conference (SPLC).* Springer, 2010, pp. 498–499.

[8] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. “Variability Modeling in the Real: A Perspective From the Operating Systems Domain”. In: *Proc. IEEE/ACM Int’l Conf. on Automated Software Engineering (ASE ’10).* ACM, 2010, pp. 73–82.

[9] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Boston, MA, USA: Addison-Wesley, 2001.

[10] K. Czarnecki and U. W. Eisenecker. *Generative Programming.* Boston, MA, USA: Addison-Wesley, 2000.

[11] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. “An Exploratory Study of Cloning in Industrial Software Product Lines”. In: *Proc. European Conf. on Software Maintenance and Reengineering (CSMR ’13).* IEEE, 2013, pp. 25–34.

[12] E. van Emden and L. Moonen. “Java Quality Assurance by Detecting Code Smells”. In: *Proc. Working Conf. on Reverse Engineering (WCRE ’02).* IEEE, 2002, pp. 97–106.

[13] M. D. Ernst, G. J. Badros, and D. Notkin. “An Empirical Analysis of C Preprocessor Use”. In: *IEEE Trans. Softw. Eng.* 28.12 (2002), pp. 1146–1170.

[14] D. Faust and C. Verhoef. “Software product line migration and deployment”. In: *Softw.: Pract. Exper.* 33.10 (2003), pp. 933–955.

[15] J.-M. Favre. “Understanding-in-the-Large”. In: *Proc. Int’l Work. on Program Comprehension (IWPC ’97)*. IEEE, 1997, pp. 29–38.

[16] J. Feigenspan, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. “Measuring Programming Experience”. In: *Proc. Int’l Conf. on Program Comprehension (ICPC ’12)*. 2012, pp. 73–82.

[17] W. Fenske and S. Schulze. “Code Smells Revisited: A Variability Perspective”. In: *Proc. Int’l Work. on Variability Modeling of Software-Intensive Systems (VaMoS ’15)*. ACM, 2015, pp. 3–10.

[18] W. Fenske, S. Schulze, D. Meyer, and G. Saake. “When Code Smells Twice as Much: Metric-Based Detection of Variability-Aware Code Smells”. In: *Proc. Int’l Working Conf. on Source Code Analysis and Manipulation (SCAM ’15)*. IEEE, 2015, pp. 171–180.

[19] W. Fenske, S. Schulze, and G. Saake. “How Preprocessor Annotations (Do Not) Affect Maintainability: A Case Study on Change-Proneness”. In: *Proc. Int’l Conf. on Generative Programming: Concepts & Experiences (GPCE ’17)*. ACM, 2017, pp. 77–90.

[20] M. Fowler, K. Beck, J. Brant, and W. Opdyke. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley, 1995.

[22] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. “Do Code Clones Matter?” In: *Proc. Int’l Conf. on Software Engineering (ICSE ’09)*. IEEE, 2009, pp. 485–495.

[23] C. Kapser and M. W. Godfrey. “”Cloning Considered Harmful” Considered Harmful: Patterns of Cloning in Software”. In: *Empir. Softw. Eng.* 13.6 (2008), pp. 645–692.

[24] C. Kästner and S. Apel. “Virtual Separation of Concerns – A Second Chance for Preprocessors”. In: *J. Object Technol.* 8.6 (2009), pp. 59–78.

[25] C. Kästner, S. Apel, and M. Kuhlemann. “Granularity in Software Product Lines”. In: *Proc. Int’l Conf. on Software Engineering (ICSE ’08)*. ACM, 2008, pp. 311–320.

[26] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Upper Saddle River, NJ, USA: Prentice-Hall, 1978.

[27] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol. “An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness”. In: *Empir. Softw. Eng.* 17.3 (2012), pp. 243–275.

[28] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. “Aspect-Oriented Programming”. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer, 1997, pp. 220–242.

[29] B. Kitchenham. “Guidelines for performing systematic literature reviews in software engineering”. In: *EBSE Technical Report*. 2007.

[30] R. Koschke, P. Frenzel, A. P. J. Breu, and K. Angstmann. “Extending the Reflexion Method For Consolidating Software Variants Into Product Lines”. In: *Softw. Qual. J.* 17.4 (2009), pp. 331–366.

[31] J. Krinke. “Is cloned code more stable than non-cloned code?” In: *Proceedings of the Working Conference on Source Code Manipulation and Analysis (SCAM)*. IEEE, 2008, pp. 57–66.

[32] C. W. Krueger. “Easing the Transition to Software Mass Customization”. In: *Proc. Int’l Work. on Software Product-Family Engineering (PFE ’01), Revised Papers*. Springer, 2002, pp. 282–293.

[33] M. M. Lehman. “Programs, Life Cycles, and Laws of Software Evolution”. In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076.

[34] V. I. Levenshtein. “Binary Codes Capable of Correcting Deletions, Insertions, and Reversals”. In: *Soviet Physics Doklady* 10.8 (1966), pp. 707–710.

[35] Y. Li, S. Schulze, and G. Saake. “Extracting Features from Requirements: Achieving Accuracy and Automation with Neural Networks”. In: *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 477–481.

[36] Y. Li, S. Schulze, and G. Saake. “Reverse Engineering Variability from Natural Language Documents: A Systematic Literature Review”. In: *Proceedings of the International Systems and Software Product Line Conference (SPLC)*. ACM, 2017, pp. 133–142.

[37] Y. Li, S. Schulze, and G. Saake. “Reverse Engineering Variability from Requirement Documents based on Probabilistic Relevance and Word Embedding”. In: *Proceedings of the International Systems and Software Product Line Conference (SPLC)*. ACM, 2018, pp. 121–131. URL: http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/LiSS+SPLC2018.pdf.

[38] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. “An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines”. In: *Proc. Int’l Conf. on Software Engineering (ICSE ’10)*. Cape Town, South Africa: ACM, 2010, pp. 105–114.

[39] J. Liebig, C. Kästner, and S. Apel. “Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code”. In: *Proc. Int’l Conf. on Aspect-Oriented Software Development (AOSD ’11)*. ACM, 2011, pp. 191–202.

[40] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. "A Quantitative Analysis of Aspects in the eCos Kernel". In: *Proc. ACM SIGOPS/EuroSys European Conf. on Computer Systems (EUROSYS '11).* ACM, 2006, pp. 191–204.

[41] R. Malaquias, M. Ribeiro, R. Bonifácio, E. Monteiro, F. Medeiros, A. Garcia, and R. Gheyi. "The Discipline of Preprocessor-Based Annotations Does #ifdef TAG n't #endif Matter". In: *Proc. Int'l Conf. on Program Comprehension (ICPC '17).* IEEE, 2017, pp. 297–307.

[42] J. D. McGregor, L. M. Northrop, S. Jarrad, and K. Pohl. "Initiating Software Product Lines". In: *IEEE Softw.* 19.4 (2002), pp. 24–27.

[43] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. "The Love/Hate Relationship with the C Preprocessor: An Interview Study". In: *Proc. European Conf. on Object-Oriented Programming (ECOOP '15).* Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015, pp. 495–518.

[44] F. Medeiros, M. Ribeiro, and R. Gheyi. "Investigating Preprocessor-Based Syntax Errors". In: *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE '13).* ACM, 2013, pp. 75–84.

[45] F. Medeiros, M. Ribeiro, R. Gheyi, S. Apel, C. Kästner, B. Ferreira, L. Carvalho, and B. Fonseca. "Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell". In: *IEEE Trans. Softw. Eng.* 44.5 (2018), pp. 453–469.

[46] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, and R. Gheyi. "An Empirical Study on Configuration-Related Issues: Investigating Undeclared and Unused Identifiers". In: *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE '15).* ACM, 2015, pp. 35–44.

[47] J. Melo, C. Brabrand, and A. Wąsowski. "How Does the Degree of Variability Affect Bug Finding?" In: *Proc. Int'l Conf. on Software Engineering (ICSE '16).* ACM, 2016, pp. 679–690.

[48] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur. "DECOR: A Method for the Specification and Detection of Code and Design Smells". In: *IEEE Trans. Softw. Eng.* 36.1 (2010), pp. 20–36.

[49] R. Muniz, L. Braz, R. Gheyi, W. Andrade, B. Fonseca, and M. Ribeiro. "A Qualitative Analysis of Variability Weaknesses in Configurable Systems with# ifdefs". In: *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS).* ACM, 2018, pp. 51–58.

[50] D. L. Parnas. "On the Design and Development of Program Families". In: *IEEE Trans. Softw. Eng.* 1 (1976), pp. 1–9.

[51] D. L. Parnas. "Software Aging". In: *Proc. Int'l Conf. on Software Engineering (ICSE '94).* IEEE, 1994, pp. 279–287.

[52] K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Berlin Heidelberg, Germany: Springer, 2005.

[53] C. Prehofer. "Feature-Oriented Programming: A Fresh Look at Objects". In: *Proc. European Conf. on Object-Oriented Programming (ECOOP '97).* Springer, 1997, pp. 419–443.

[54] D. Romano and M. Pinzger. "Using Source Code Metrics to Predict Change-Prone Java Interfaces". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '02).* IEEE, 2011, pp. 303–312.

[55] C. D. Rosso and C. Riva. "Experiences with Software Product Family Evolution". In: IEEE, 2003, p. 161. DOI: doi.ieeecomputersociety.org/10.1109/IWPSE.2003.1231223.

[56] J. Rubin and M. Chechik. "A Framework for Managing Cloned Product Variants". In: *Proc. Int'l Conf. on Software Engineering (ICSE '13).* San Francisco, CA, USA: IEEE, 2013, pp. 1233–1236.

[57] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. "Delta-Oriented Programming of Software Product Lines". In: *Proc. Int'l Software Product Line Conf. (SPLC '10).* Springer, 2010, pp. 77–91.

[58] A. Schlie, S. Schulze, and I. Schaefer. "Comparing Multiple MATLAB/Simulink Models Using Static Connectivity Matrix Analysis". In: *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2018, pp. 185–196.

[59] A. Schlie, D. Wille, S. Schulze, L. Cleophas, and I. Schaefer. "Detecting Variability in MATLAB/Simulink Models: An Industry-Inspired Technique and Its Evaluation". In: *Proceedings of the International Systems and Software Product Line Conference (SPLC).* ACM, 2017, pp. 215–224.

[60] S. Schulze and W. Fenske. "Analyzing the Evolution of Preprocessor-Based Variability: A Tale of a Thousand and One Scripts". In: *Proc. Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM '18).* IEEE, 2018, pp. 50–55.

[61] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. "Reverse Engineering Feature Models". In: *Proceedings of the International Conference on Software Engineering (ICSE).* ACM, 2011, pp. 461–470.

[62] Y. Smaragdakis and D. Batory. "Mixin Layers: An Object-oriented Implementation Technique for Refinements and Collaboration-based Designs". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11.2 (2002), pp. 215–255.

[63] *Software Engineering - Product Quality, ISO/IEC 9126-1.* Tech. rep. International Organization for Standardization, 2001.

[64] H. Spencer and G. Collyer. “#ifdef Considered Harmful, or Portability Experience With C News”. In: *Proc. Proc. USENIX Conf.* USENIX Association, 1992, pp. 185–197.

[65] Ş. Stănciulescu, S. Schulze, and A. Wąsowski. “Forked and Integrated Variants in an Open-Source Firmware Project”. In: *Proc. Int’l Conf. on Software Maintenance and Evolution (ICSME ’15).* IEEE, 2015, pp. 151–160.

[66] M. VanHilst and D. Notkin. “Using Role Components in Implement Collaboration-based Designs”. In: *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).* ACM, 1996, pp. 359–369.

[67] D. Wille, S. Schulze, C. Seidl, and I. Schaefer. “Custom-Tailored Variability Mining for Block-Based Languages”. In: *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 2016, pp. 271–282.

Hiermit erkläre ich, dass ich die Habilitationsschrift und die im Verzeichnis meiner wissenschaftlichen Veröffentlichungen angegebenen Werke selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 19.03.2019