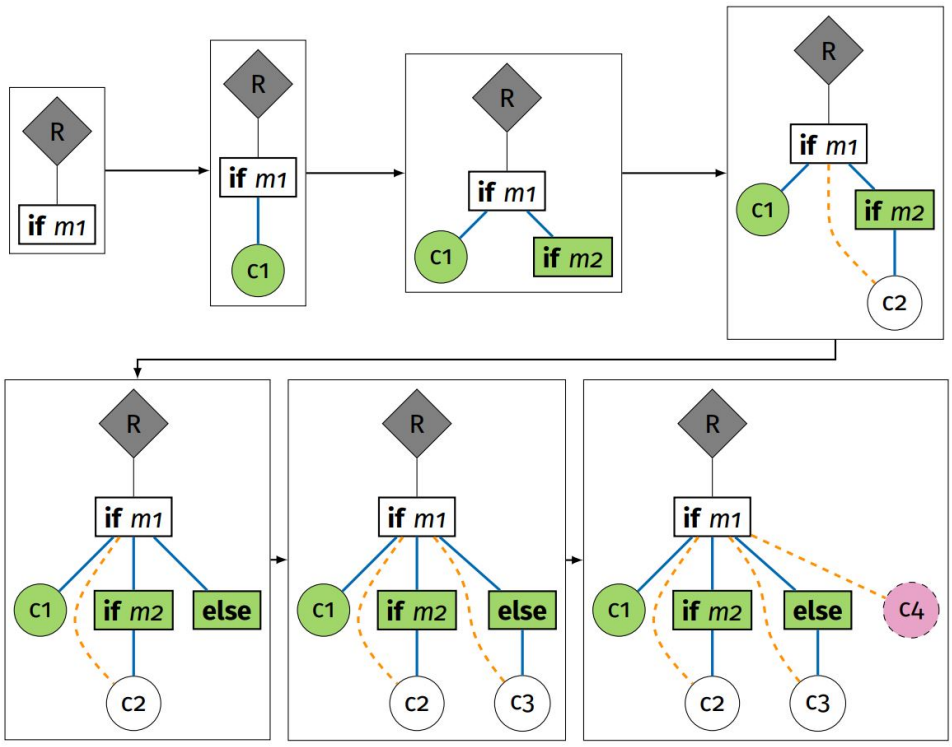
Software Engineering
Programming Languages

ulm university universität
uulm



Empirical Evaluation of Feature Trace Recording on the Edit History of Marlin

Sören Viegener

Bachelor's Thesis

Institute of Software Engineering and Programming Languages

Sören Viegener:
Empirical Evaluation of Feature Trace Recording on the Edit History of Marlin

Advisor:

Prof. Dr. Thomas Thüm
M.Sc. Paul Maximilian Bittner
Institute of Software Engineering and Programming Languages

Bachelor's Thesis, University of Ulm, 2021.

Abstract

Between software product lines and clone-and-own software development lies a large spectrum of approaches for software projects. Feature trace recording is a novel method for tracking feature mappings during the development of a clone-and-own project. An existing evaluation shows that feature trace recording is able to perform edits usually found in a software product line but does not give any insight into its applicability to real software development. This thesis extends this evaluation by empirically evaluating feature trace recording on the commit history of *Marlin*, an open-source preprocessor-based software product line. We gather empirical data by categorizing edits into edit patterns. From the pattern matches, we reverse engineer how these edits could be reproduced using feature trace recording (i.e., which feature context is necessary). When using the edit patterns presented by Stănciulescu et al., we discovered problems regarding ambiguity, mutual exclusivity, and exhaustion of the patterns. To solve these problems, we refine and extend the patterns introducing a new classification of edit patterns together with a mechanism to detect them. With this new classification, we present exact definitions for edit patterns that are mutually exclusive and exhaustive regarding all edited lines of code. We implemented a tool for edit pattern detection and to perform the reverse engineering of edits automatically. The results of our evaluation include exact amounts of all pattern matches found in the commit history of *Marlin*. From our reverse engineering of the feature contexts, we can conclude that changing the feature context may not be a large effort for developers. We also conclude that the complexity of feature contexts may be reasonably low, and we provide first observations on when the feature context can be omitted and the similarity of feature context and target feature mapping.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

When working on software projects with a large amount of variability, software product lines are the software engineering method of choice. When developing different variants of software that share common artifacts, software product lines yield many advantages, such as enhanced quality of code, reduced development costs, and shorter time to market [Kru06], [Nor08], [PBvdL05], [WL99]. However, because of the great initial investment needed for software product lines and because for some projects not all requirements are known at the beginning of development, ad-hoc solutions, such as clone-and-own, are commonly used in practice [DRB+13]. Clone-and-own is a term used to describe the development of different variants (i.e., clones) of software in parallel. When a new requirement creates the need for a new variant, an existing variant is cloned and modified. Clone-and-own is simple but effective when there are only a few variants, but becomes cumbersome, when the amount of variants increases [AJB+14], [DRB+13], [RCC13].

For clone-and-own software, there is often a point at which support for new variants or the maintenance of existing clones is not feasible anymore [DRB+13], [WSSS16]. Thus, the migration of software projects to software product lines has been a focus of research [FMS+17], [KDO14], [KFBA09], [LC13], [WSSS16]. These migrations usually require feature mappings (i.e., implementations assigned to the features they implement) which need to be recovered retroactively as they are typically not documented in a clone-and-own project. As developers might not know anymore, which implementation belongs to which feature, the migrations either involve a large initial effort or work on the basis of incorrect feature traces. They are thus not a simple solution to the problem.

Between full software product lines and clone-and-own projects lies a large spectrum of different software engineering methods [KB20]. One intermediate state, which is proposed and discussed in current research [JBAC15], [FLLHE15], [LBG17], is a clone-and-own project with a targeted synchronization between

clones. Differences and commonalities between clones are assessed in terms of features, known from software product-line engineering. Changes to one clone are transferred to other clones containing the same feature that was modified. The goal of this method is to avoid duplicate work because bug fixes or new requirements concerning more than one variant can be synchronized. Before any synchronization or migration can take place, knowledge about the location of a feature's implementation (i.e., feature traces or feature mappings) is needed.

Feature trace recording is a new method of tracking the implementations of features during the development of a software project [BST+]. When writing code for a specific variant, the developer specifies the features they are currently working on as a propositional formula called the feature context. When editing source code, feature mappings are recorded for the edited artifacts depending on the feature context, the type of the performed edit (i.e., insertion, deletion, move, or update), and the previous feature mappings. The feature context is explicitly allowed to be unspecified (i.e., set to a null value) which accounts for old or new code without knowledge of the implemented features.

As feature trace recording is in the early stages of research, there is only a theoretical evaluation using edit patterns described by Stănciulescu et al. [SBWW16], which demonstrates that feature trace recording is capable of handling common edits performed in software product lines [BST+]. This evaluation does not offer precise information regarding the applicability to real software development. To evaluate this applicability, empirical data is needed. Using the edit history of a clone-and-own project would require manually identifying feature traces to use as ground truth. This would be too time consuming or might produce incorrect feature traces. This is why throughout research, such empirical data is usually gathered by analyzing an existing software product line. Product lines already contain feature traces which can serve as a ground-truth for evaluation [SBWW16], [SSW15].

The goal of this thesis is the empirical evaluation of feature trace recording by using data gathered from the software product line *Marlin* [vdZ]. The commit history of *Marlin* will be analyzed to find edit patterns. For this, we refine and extend the edit patterns described by Stănciulescu et al. [SBWW16] which were used in the theoretical evaluation of feature trace recording [BST+]. These edit patterns mainly revolve around the addition or removal of source code lines surrounded by C preprocessor annotations which identify the code's features. We present a new classification of edit patterns, give exact definitions and describe how we are able to match these patterns in the commit history of *Marlin*. As our new classification enables the patterns to be mutually exclusive and exhaustive regarding all edited lines of code, we are able to match every edited line in the *Marlin* repository to exactly one of these edit patterns.

Each edit pattern contains at least one feature mapping which is a formula over features found in the C preprocessor annotations. From a pattern match with its

feature mapping, the feature context that would have been necessary to perform the exact same change using feature trace recording in a variant of the software project can be reverse engineered. This was presented in the theoretical evaluation of feature trace recording [BST+]. With information about the occurrences of the different edit patterns and the feature contexts, we evaluate the applicability of feature trace recording to real software projects. The evaluation aims to answer several research questions which range from the diversity of necessary feature contexts in a single commit to the complexity of feature contexts. We present and discuss these research questions in the evaluation in Chapter 5. With answers to these questions, the existing theoretical evaluation of feature trace recording [BST+] is thus expanded by an empirical evaluation of the subject.

To perform this evaluation automatically, we implemented an evaluation tool, called *DiffDetective*. DiffDetective is able to take a repository of a software product line as input and detect edit patterns in its commit history. Changes in commits are automatically split by file into patches which each consist of all the changes performed on a single file in a single commit. The tool then analyzes each patch with regard to the edit patterns. This is done by finding occurrences of the edit patterns and also detecting the corresponding feature mappings. With the pattern match and the feature mapping, DiffDetective can reverse engineer possible feature contexts. The data gathered by DiffDetective thus consists of all pattern matches found throughout the commit history, the feature mappings they contain, and the reverse-engineered feature contexts. The main data set for the evaluation is the *Marlin* repository[1], as it is commonly used throughout research [KGS+18], [SSW15] and was also used by Stănciulescu et al. [SBWW16].

DiffDetective enables us to present precise amounts for all pattern matches in the commit history of *Marlin*. We additionally present the number of patches affected by each pattern and the number of lines of code that are matched to each pattern. By reverse-engineering feature contexts from pattern matches, we provide the first empirical results for the usage of feature trace recording in practice. From the number of feature contexts per commit and the number of changed lines of code per feature context, we conclude that specifying the feature context may not be a large effort for developers on average. Evidence is also given by the mostly low complexity of feature contexts which we found throughout all commits. Furthermore, our results provide first insights into how often the feature context can be omitted and the similarity of feature context and target feature mappings.

This thesis is structured as follows. In Chapter 2, we explain the basics of software product lines, clone-and-own development, and the git diff notation which is used throughout this thesis. In Chapter 3, we describe the design and requirements for the evaluation, including the edit patterns and how we detect them.

[1] https://github.com/MarlinFirmware/Marlin

This lays the foundations for Chapter 4, where we present and discuss the implementation and structure of DiffDetective. We present and analyze the results of the evaluation in Chapter 5. We also introduce and answer our research questions there. In Chapter 6, we put this thesis into context of other current research on related topics. The final chapter, Chapter 7 offers a summary of the insights of this thesis and an outlook into possible future tasks continuing this research.

2. Background

In this chapter, we present background information relevant to this thesis. We will briefly explain two important concepts: Software product-line engineering and clone-and-own development. For our evaluation, we use the data gathered from a software product line to simulate clone-and-own with additional feature mappings. Knowledge of the two concepts is thus required to understand the subject of this thesis. We also provide an introduction to the *git diff* notation as we use this notation to describe the edits and edit patterns we present.

2.1 Software Product-Line Engineering

Software product-line engineering (SPLE) is a software engineering technique in which a group of software, a software product line, is defined by *features* [WL99], [PBvdL05]. A feature is a specific requirement of the software and can be defined with any granularity. A combination of these features is called a *configuration* with which a specific software variant can be generated from an integrated codebase. A software product line thus enables building several different *variants* of the same software from a selection of features. To demonstrate the concept, we provide an example of a product line. In Figure 2.1 you can see the features of a *Car* product line. The features are structured in a *feature model* which shows

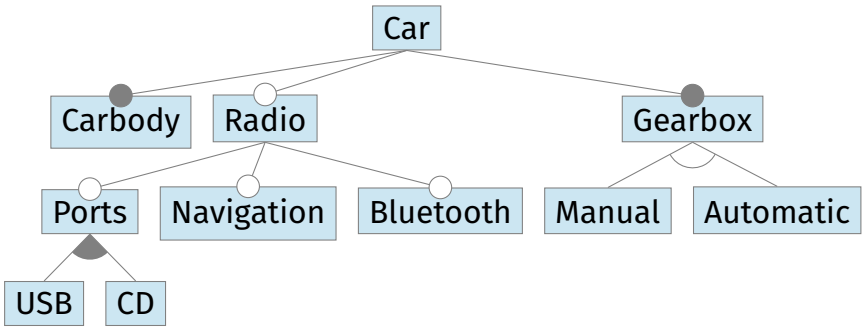


Figure 2.1: A Feature Model of a Product Line for a Car

the dependencies of the different features. In this case, for instance, *Navigation* could only be in a configuration which also contains *Radio*. A configuration for this product line could thus consist of the following features: *Carbody*, *Radio*, *Navigation*, *Gearbox*, *Manual*. From this configuration, a specific variant can be built.

To generate a variant from a configuration of features, these features need to be implemented. A *feature trace* is the connection of a feature to its implementations in the code. We call the inverse relation, which is the connection of a specific code artifact to the feature it implements, *feature mapping*. In our thesis, we work with software product lines that use the C preprocessor for defining feature mappings [KA08]. A feature mapping for a block of code can look like this:

```
#if Bluetooth && Navigation
  /* block of code */
#endif
```

The block of code is enclosed in C preprocessor annotations and will thus only be included in the program when the condition of the **`#if`**-annotation evaluates to true. The code thus has the feature mapping *Bluetooth* $\wedge$ *Navigation* and will only be present in a variant with a configuration that includes both *Bluetooth* and *Navigation*. As these preprocessor annotations can be nested, we refer to the combination of all surrounding feature mappings of a block of code as the *presence condition*.

```
#if Manual
  /* block of code (c1) */
  #if Bluetooth && Navigation
    /* block of code (c2) */
  #endif
#endif
```

The feature mappings in the example are *Manual* for `c1` and *Bluetooth* $\wedge$ *Navigation* for `c2`. As `c2` is wrapped by both annotations, its presence condition is *Manual* $\wedge$ *Bluetooth* $\wedge$ *Navigation*. The code will thus only be present in a variant with a configuration that includes *Manual*, *Bluetooth*, and *Navigation*.

2.2 Clone-and-Own Development

Clone-and-own is a broad term used to describe software projects where existing software is cloned and modified to fulfill new requirements [AJB+14], [RCC13], [DRB+13]. These clones of software can be used to develop different *variants*. For a small number of variants, clone-and-own provides a lot of flexibility and is easily maintainable. When, however, many new variants are required, maintaining them is often cumbersome. In such a project, feature traces are usually not available but could possibly simplify development. Feature trace recording

extends regular clone-and-own development as it aids in tracking feature traces during the development of a clone-and-own project. The goal of this approach is to eventually synchronize changes to feature implementations between different clones by assigning clones a configuration.

2.3 Git Diff Notation

The *git diff* notation is a way of representing line-based changes of a text-based file [THCL]. It is used by the version control system *git* and will also be used throughout this thesis. For this notation, the first character in each line defines whether this line was added ('+'), removed ('-'), or remained unchanged (no character). To better visualize changes, we color removed lines in pink and added lines in green. As seen in Figure 2.2, modifying an existing line is modeled as a removal and an addition.

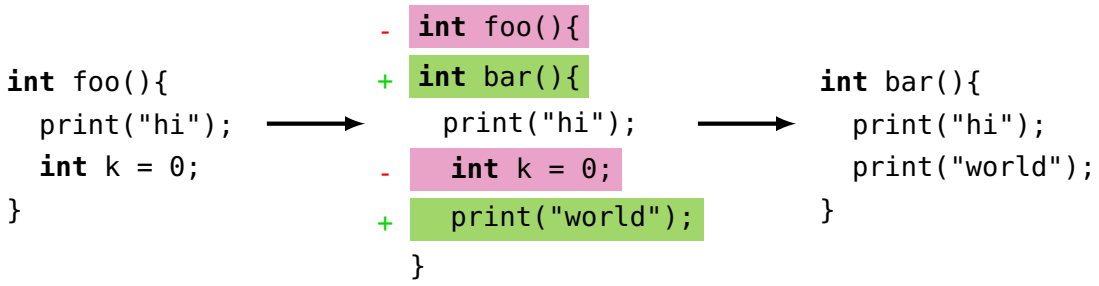


Figure 2.2: A Method Before a Change, The Diff of the Change in *git diff* Notation, and the Method After the Change

3. Classification and Detection of Edit Patterns

This chapter presents the main concept of this thesis including the methods used for our evaluation. We describe feature trace recording and its existing evaluation [BST+] and present the edit patterns which are one of the contributions of our work and the basis of our evaluation.

To evaluate feature trace recording, we categorize changes made in the commit history of a software product line by edit patterns. Initially, we tried to use the patterns presented by Stănciulescu et al. [SBWW16]. During our work, we discovered several problems, including ambiguous matches and the detection of nested patterns, because of which we had to refine and extend the patterns. We thus present a new classification of edit patterns: *atomic patterns* and *semantic patterns*. Atomic patterns describe basic, mutually exclusive code changes and can be used to evaluate all insertions and deletions of code and feature mappings as they are designed to be exhaustive. Semantic patterns are composed of atomic patterns and thus describe larger changes that hold some semantic value. They are not exhaustive and may overlap but can be used to gain more insight into the actual code changes performed by a developer. For our evaluation, we detect these patterns in the commit history of the *Marlin* repository and use the data gathered to reverse engineer feature contexts used for feature trace recording.

In Section 3.1, we describe the main concepts of feature trace recording. We explain how we expand the already existing theoretical evaluation of feature trace recording in Section 3.2. We introduce the edit patterns and how they can be detected in Section 3.3. In Section 3.4, we then conclude the chapter with a summary.

3.1 Feature Trace Recording

Feature trace recording is a novel method for tracking feature-to-code mappings (i.e., feature traces or feature mappings) during development of a clone-and-own software project (i.e., a set of multiple software variants). When working on a variant, the developer can specify the feature they are currently working on with the *feature context*. With the edit performed by the developer and the feature context, feature trace recording can calculate the feature mappings for the edited artifacts. An important part of feature trace recording is the option to specifically not set the feature context (i.e., set it to `null`), for example when the developer does not know the current feature. As features that are not present in the current variant's configuration can not be implemented in the variant, the feature context, when set, is limited to formulas that evaluate to true under the variant's configuration [BST+].

Feature trace recording supports four different edit operations so far that are performed on an abstract syntax tree (AST). *Insertion* covers the addition of new code (i.e., adding nodes to the AST) and is the most common edit operation together with *deletion* which is the removal of already existing code (i.e., removing nodes from the AST). *Update* and *move* cover changes to metadata (e.g., names) of existing nodes in the AST and moving subtrees from one location to another, respectively. Any code change performed by a developer can be described with a combination of these operations as any operation can be described using insertions and deletions. If no grammar to construct the AST for the target language is available, an AST can be built using single lines in the code as nodes This enables feature trace recording to work with any programming language and textual file.

The core of feature trace recording is an algorithm that calculates the feature mapping for each edited node in the AST. For this, the edit operation that was performed, the feature context that was specified, the old AST, and the old feature mapping are used as input. The output consists of the new feature mapping for all edited nodes. A complete description of the algorithm can be found in the original work [BST+].

3.2 Extending the Evaluation of Feature Trace Recording

The existing theoretical evaluation of feature trace recording demonstrates that the algorithm can handle any changes usually performed in preprocessor-based software product lines [BST+]. This evaluation is based on the edit patterns presented by Stănciulescu et al. which describe different edits found in the history of a preprocessor-based software product line [SBWW16]. They present 14 different edit patterns divided into three categories: code-adding patterns, code-removing patterns, and other patterns. The patterns use the *git diff* notation

explained in Section 2.3.

An example for a code-adding pattern is the *AddIfdefWrapElse* pattern which describes the addition of an **#if**-**#else**-**#endif**-block where the **#if**-branch contains newly added code and the **#else**-branch wraps existing code:

```
+ #if m
+   /* inserted code (c1) */
+ #else
    /* existing code (c2) */
+ #endif
```

The existing evaluation shows how this pattern can be reproduced using feature trace recording. For this, the feature context can be set to the feature mapping *m*, the formula after the **#if**-annotation. Using this feature context in a single variant, the existing code (c2) can be removed and the new code (c1) can be added to achieve the desired result.

RemNormalCode is an example for a a code-removing pattern. It describes the removal of code without removing possible surrounding preprocessor annotations:

```
  #if m
-   /* removed code */
  #endif
```

This edit pattern can be recreated with feature trace recording in a single variant. The feature context can be set to `true`. If there is a surrounding annotation with a feature mapping, like in the above pattern, the feature context can also be set to *m*, any formula weaker than *m*, or even to `null`.

For feature trace recording, only 8 of the 14 patterns were relevant for the existing evaluation. The other patterns describe changes that are an orthogonal concern to feature trace recording, as feature trace recording is intended to only work with changes of code and not with direct changes of feature mappings which could be performed using other tools. These other patterns comprise changing feature mappings directly and repairing syntactically ill-formed annotations. The evaluation shows that each of the relevant patterns can be recreated using feature trace recording. This means that feature trace recording is able to perform any relevant code change usually found in a preprocessor-based software product line. The existing evaluation mainly focused on the general applicability of these edit patterns and did not evaluate specific edits occurring in real software projects. The results suggest that feature contexts are sometimes simpler than the corresponding feature mappings and that the feature context potentially needs to be switched less often than a direct specification of feature mappings would require [BST+].

The goal of this thesis is to extend this evaluation by answering several research questions regarding the applicability of feature trace recording to real software projects using empirical data from a software product line. The insights and methods of the existing theoretical evaluation are the starting point of our work. While matching the existing edit patterns of Stănciulescu et al. [SBWW16], we encountered several problems. First of all, we discovered that the descriptions of the patterns are ambiguous as the authors did not provide the exact definitions of the patterns. This made it difficult for us to reproduce their results. Second, the patterns are sometimes overlapping as noted by the authors. As the patterns are also not exhaustive they may not cover all possible edits. We can thus not use the existing descriptions to exactly categorize all edits in a software project.

For our evaluation, we refine and extend the edit patterns which we divide into two larger categories. *Atomic patterns* describe changes to single lines of code. We identify a total of 7 of these patterns which are based on the patterns of Stănciulescu et al. They are mutually exclusive and each edited line of code can be categorized into exactly one of these patterns which enables us to unambiguously classify edits. *Semantic patterns* categorize larger edits usually made up of multiple atomic patterns. These patterns are mostly patterns presented by Stănciulescu et al. and hold some semantic value that emerges from specific compositions of atomic patterns. The semantic patterns are neither necessarily mutually exclusive nor exhaustive but can be used to get better insight into actual edits performed by developers. We will describe all edit patterns and how we detect them in detail in Section 3.3.

As ground truth data for our evaluation we use the *Marlin* repository[1]. As this repository currently consists of 15,749 commits and the revision used by Stănciulescu et al. already had 3747 commits [SBWW16], a manual analysis of each commit is not feasible. We thus implemented an evaluation tool, called DiffDetective, to perform the edit pattern detection and parts of our evaluation automatically. We describe the implementation of DiffDetective in detail in Chapter 4.

3.3 Edit Pattern Detection

To detect edit patterns in the complete commit history of *Marlin*, we aimed to recreate the results of Stănciulescu et al. and thus initially used the same procedure as they did. We first separate each commit into patches. A patch contains the complete changes performed on a single file in a single commit. We discard merge commits as they do not describe the direct modification of source code. They are a concern that could be evaluated in future works on this topic. We then filter the list of patches to only contain modifications performed on the source files of the software project. This means removing all patches concerning the documentation or external files such as Arduino library files. We are then able to classify the edits made in each patch by matching edit patterns.

[1] https://github.com/MarlinFirmware/Marlin

Stănciulescu et al. detected the patterns using regular expressions and a parsing library [SBWW16] (while the parsing library is not mentioned in the original paper, the authors told us about it when contacting them). Initially, we also tried using regular expressions to detect these patterns but discovered some problems with this approach. Regular expressions cannot model languages with recursive structures. In the case of edit patterns, preprocessor annotations can be nested which can not be detected using just regular expressions. We thus discarded this option for edit pattern detection.

To simplify the pattern detection, we first looked at different ways of representing a patch. An initial approach consisted of parsing the patch to create a tree representing it. The nodes of the tree were then either blocks defined by preprocessor annotations or lines of code. For most patches, this representation would suffice, but we discovered that not all types of differences can be modeled using just a single tree. Extending this tree, we built a representation using a directed acyclic graph which we call *diff tree*. This representation can be built from a patch and we use it to detect the edit patterns. In the following sections, we present an algorithm to build this diff tree, our new edit patterns, and how we detect them. We also show how we use the edit patterns to evaluate trace recording.

3.3.1 Step 1: Diff Tree Construction

In the first phase, we create a diff tree for each patch. The diff tree is a directed acyclic graph. It contains line-based changes of source code with subtrees representing preprocessor annotation scopes. Each node in the diff tree is categorized using a *code type* and a *diff type*. The code type specifies whether the node represents an `#if`-block (can be either `#if`, `#ifdef`, or `#ifndef`), `#elif`-block, `#else`-block, or a line of code. We call these code types **if**, **elif**, **else**, and **code**, respectively. We also refer to nodes with code type **code** as *code nodes* and other nodes as *annotation nodes*. The diff type specifies whether the corresponding code lines or annotations were added, removed, or remained unchanged in the patch. We call the diff types **add**, **remove**, and **none**, respectively. Every node has at most two parents, the only node without parents being the root node. The root node represents the complete patch and always has diff type **none** and code type **if** with the feature mapping `true`. Nodes of diff type **none** have two parents, an **after parent** and a **before parent**. The **after parent** identifies the surrounding preprocessor block in the file after the patch. The **before parent** identifies the surrounding preprocessor block in the file before the patch. Nodes of diff type **add** only have an **after parent**, nodes of diff type **remove** only have a **before parent**. The **after parent** of a node can never have diff type **remove** and the **before parent** can never have diff type **add**.

As a patch contains all information about a file before and after a change, we can build the complete diff tree from it. To do this, we present Algorithm 1. The algorithm uses two stacks to keep track of the current **before parent** and **after parent**. The *before stack* always contains the currently innermost annotation

Algorithm 1 Building a Diff Tree From a Patch

Input: a patch

1: initialize *before stack* with *root node*
2: initialize *after stack* with *root node*
3:
4: **for all** lines in the patch **do**
5: $\delta \leftarrow$ identify diff type
6: $\gamma \leftarrow$ identify code type
7: $\sigma \leftarrow$ get relevant stacks using δ
8:
9: **if** γ = `endif` **then**
10: pop σ until γ = `if` is popped
11: **else**
12: create *new node* with δ, γ and parents from σ
13: **if** $\gamma \neq$ `code` **then**
14: push *new node* to σ
15: **end if**
16: **end if**
17: **end for**

Output: all nodes of the diff tree

node in the file before the patch. The *after stack* always contains the currently innermost annotation node in the file after the patch. When the diff type δ for a line is identified, the *relevant stacks* σ for this line can be determined as follows:

$$\sigma = \begin{cases} \textit{after stack} & , \delta = \textbf{add} \\ \textit{before stack} & , \delta = \textbf{remove} \\ \textit{before} \text{ and } \textit{after stack} & , \delta = \textbf{none} \end{cases} \tag{3.1}$$

The algorithm uses an additional code type representing **`#endif`**-annotations. We call this code type **endif** and it is not contained in the final diff tree. To demonstrate how the algorithm works and to introduce how we visually represent the diff tree, we provide the following example patch:

```
  #if m1
+   /* added code (c1) */
+ #if m2
    /* code (c2) */
+ #endif
+ #else
    /* code (c3) */
+ #endif
-   /* removed code (c4) */
- #endif
```

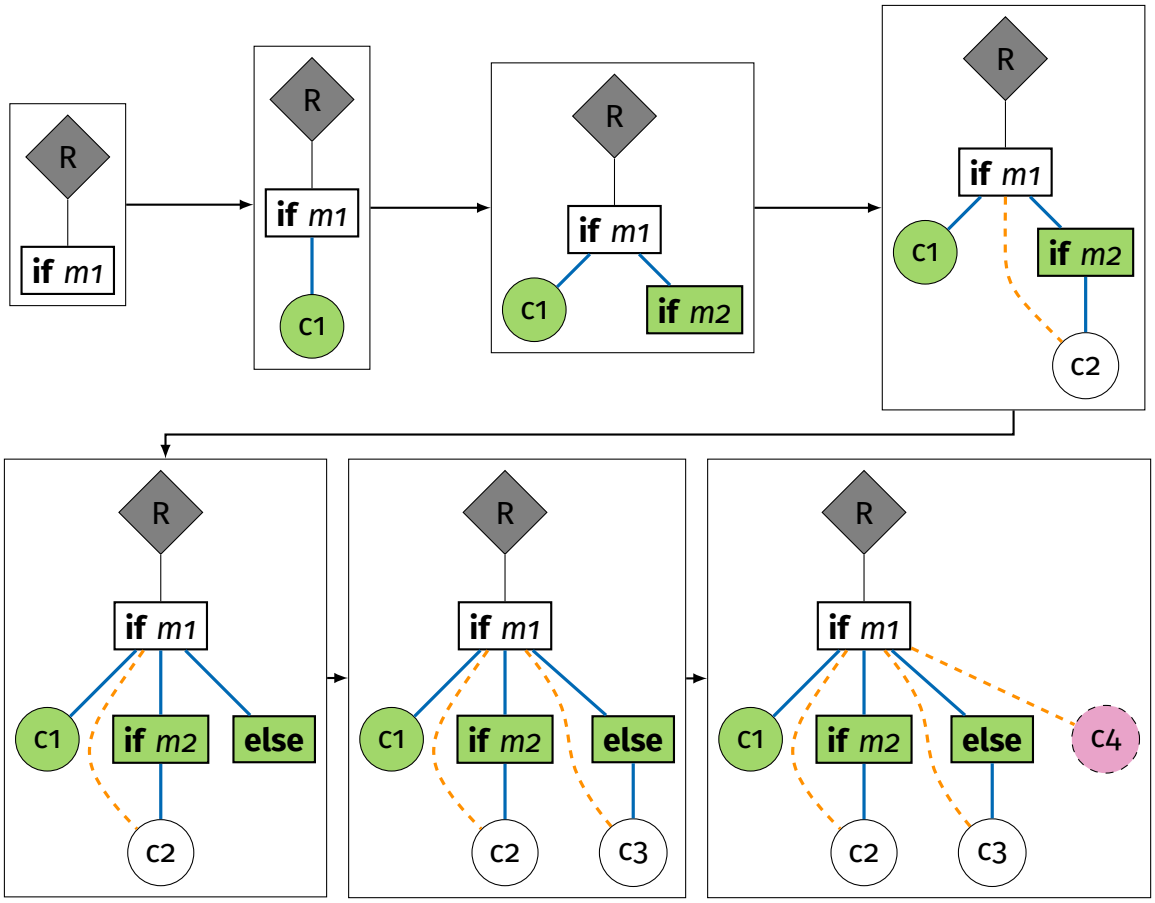


Figure 3.1: Building a Diff Tree Using Algorithm 1

In Figure 3.1, we show the diff tree at different stages of the algorithm. We start by initializing the stacks with the root node in lines 1 and 2 of the algorithm. In the diff tree, we visualize the root node in gray. The first line in the example patch is of code type **if** and diff type **none**. In line 12 of the algorithm, we add this node to the diff tree using the topmost elements of both stacks as **before** and **after parent**, respectively. As in this case both parents are the same, we only draw a single edge between the root node and the new annotation node. The new node is then pushed on both stacks in line 14 as it is present before and after the patch. The next line (c1) has code type **code** and diff type **add**. The new node is created with only a single parent being the just created annotation node (**if** *m1*). As the diff type is **add**, the parent is the **after parent** of c1. We visualize this using a blue edge in the diff tree. As the new node has code type **code**, it cannot have any children and will not be pushed to any stacks. The next line of code has code type **if** and diff type **add**. It receives the same parent (**if** *m1*) as the previous node but will be pushed to the *after stack* only, as it did not exist before the patch. This is relevant for the addition of the next code line (c2). The node of this line will be added using two different parents as the two stacks now have two different nodes on top. We visualize the **before parent** using a dashed orange edge. C2 now has two different parents as the surrounding annotations were changed in the patch. The next line, the **`#endif`**, is not added to the diff

tree but pops the *after stack* in line 10 as it has diff type **add**. The relevant stacks are popped because an `#endif` closes an annotation scope. As the stacks can also contain nodes with code type **elif** or **else**, we need to remove nodes until a node with code type **if** is removed. Next, a new node with code type **else** is added and also pushed onto the **after stack**. C3 is then added with its **after parent** being the **else** node and the node on the *before stack* (**if** *m1*) being its **before parent**. The next `#endif` then pops both annotation nodes (**else** and **if** *m1*) from the *after stack*. Then the second to last line (c4), which has code type **code** and diff type **remove** is added to the diff tree. This new node only has a **before parent** because it does not exist after the patch. The algorithm finishes by popping the *before stack* because of the final `#endif` with diff type **remove**. The final diff tree can then be seen in the last box in Figure 3.1.

3.3.2 Step 2: Analysis

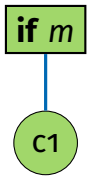
In the second step, the *analysis*, we detect edit patterns on the generated diff tree. We present two types of edit patterns: *atomic patterns* and *semantic patterns*. *Atomic patterns* are matched to each node with the code type **code**. Any code node that was changed or had its surrounding annotations changed matches exactly one atomic pattern. *Semantic patterns* are detected using the annotation nodes in the diff tree. As these patterns may match overlapping parts of the source code, a single annotation node may match multiple semantic patterns.

3.3.2.1 Detecting Atomic Patterns

We identified 7 patterns of Stănciulescu et al. to be atomic. In the following, we give exact definitions for each pattern. We changed the names of most of the atomic patterns to more accurately reflect the edit they describe. Each atomic pattern match consists of a single code node and the feature mapping of this node. In the case of code nodes with diff type **none**, the feature mappings before and after the patch are relevant. Two patterns match nodes of diff type **add** and two patterns describe the corresponding operations for nodes of diff type **remove**. The remaining three patterns describe nodes of diff type **none**. For each pattern, we explain the operation they describe and how we detect them. Note that the code and diff tree examples we give just show the relevant part of a patch for each pattern. The surrounding annotations could always additionally contain other code or annotations. When not explicitly relevant we omit the root node in our example diff trees.

AddWithMapping:
This pattern refines the *AddIfdef* pattern presented by Stănciulescu et al. but it is not limited to `#if`-annotations. The *AddWithMapping* pattern describes the addition of code that is surrounded by an `#if`, `#elif`, or `#else` annotation that is also added. An example for such a pattern can look like this:

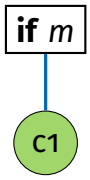
```
+ #if m
+   /* added code (c1) */
+ #endif
```



To detect the pattern, we start from a node with code type **code** and diff type **add** (c1). We then visit the parent of this node (**if** *m*) and if it is an annotation node with diff type **add** we match the *AddWithMapping* pattern. The pattern match consists of the code node (c1) and the feature mapping given by the surrounding annotation (*m*).

AddToPC:
The *AddToPC* pattern describes code that is added within an already existing presence condition. It refines the *AddNormalCode* pattern presented by Stănciulescu et al. The presence condition can also be *true*, when there is no surrounding annotation. The pattern could look like this:

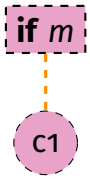
```
  #if m
+   /* added code (c1) */
  #endif
```



This pattern matches all nodes with code type **code** and diff type **add** that do not match an *AddWithMapping* pattern. We thus detect it by starting from a node with code type **code** and diff type **add** and checking whether the parent does not have diff type **add** or if it is the root node.

RemWithMapping:
This pattern describes the removal of code together with a surrounding annotation. It is the inverse of the *AddWithMapping* pattern and refines the *RemIfdef* pattern presented by Stănciulescu et al. An example for the pattern could look like this:

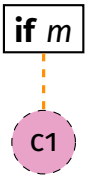
```
- #if m
-   /* removed code (c1) */
- #endif
```



To detect the pattern, we start from a node with code type **code** and diff type **remove**. We visit the parent of this node and if it is an annotation node with diff type **remove** we match a *RemWithMapping* pattern.

RemFromPC:
This pattern describes the removal of code without removing possibly surrounding annotations. It is the inverse of the *AddToPC* pattern and refines the *RemNormalCode* pattern presented by Stănciulescu et al. It could look like this:

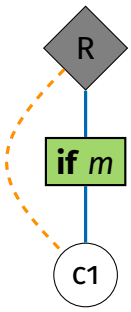
```
#if m
-   /* removed code (c1) */
#endif
```



This pattern matches all nodes with code type **code** and diff type **remove** that do not match a *RemWithMapping* pattern. We thus detect the *RemFromPC* pattern by starting from a node with code type **code** and diff type **remove** and checking whether the parent is not of diff type **remove** or if it is the root node.

WrapCode:
This pattern refines the original *WrapCode* pattern presented by Stănciulescu et al. This is the first pattern describing nodes with diff type **none**. The *WrapCode* pattern describes unchanged code that is wrapped by at least one added annotation and no removed annotation. In the example, we show code that is surrounded by a single added **#if**-annotation. Note that there could be further surrounding annotations and the **before parent** of c1 does not have to be the root node:

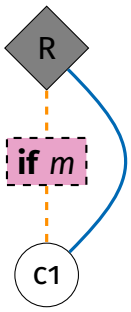
```
+ #if m
    /* unchanged code (c1) */
+ #endif
```



To detect this pattern, we start from a node with code type **code** and diff type **none**. We then look at two paths to the root node, one following the **after parent** of each node, one following the **before parent** of each node. There are two ways how this pattern can be matched. First, if the path following the **after parents** contains at least one annotation node with diff type **add** and the path following the **before parents** contains no annotation nodes with diff type **remove**, this pattern is matched. Second, when an existing **#endif** is moved, code may be wrapped without any changes to **#if** annotations. When there are no nodes with diff type **add** or **remove** in the paths to the root node, we compare the lengths of the two paths. If the path following the **after parents** is longer than the path following the **before parents** we also match this pattern because then the inspected code node has at least one additional surrounding annotation after the edit.

UnwrapCode:
This pattern refines the original *UnwrapCode* pattern presented by Stănciulescu et al. It is the inverse of the *WrapCode* pattern as it describes unchanged code that is wrapped by at least one removed annotation and no added annotation. An example could look like this. As with the example for the *WrapCode* pattern, there could be additional surrounding annotations:

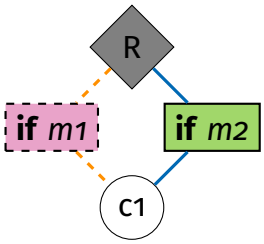
```
- #if m
    /* unchanged code (c1) */
- #endif
```



Similar to the *WrapCode* pattern, we detect it starting from a node with code type **code** and diff type **none** using the two different paths to the root node following the **before parents** or **after parents**, respectively. There are two ways how this pattern can be matched. First, if the path following the **before parents** contains at least one annotation node with diff type **remove** and the path following the **after parents** contains no annotation nodes with diff type **add**, this pattern is matched. Second, analogous to the *WrapCode* pattern, a moved `#endif` can result in unwrapped code without any changed `#if` annotation. When there are no nodes with diff type **add** or **remove** in the paths to the root node, we thus compare the lengths of the two paths. If the path following the **before parents** is longer than the path following the **after parents** we also match this pattern.

ChangePC:
This pattern refines the original *ChangePC* pattern presented by Stănciulescu et al. It is the last pattern for code nodes with diff type **none** and the final atomic pattern. It describes code for which the existing presence condition is changed. The pattern could look like this:

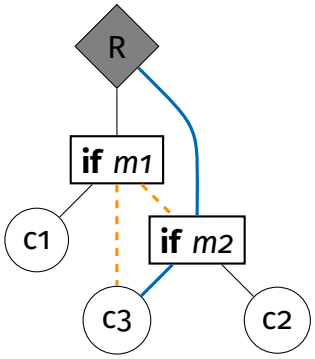
```
- #if m1
+ #if m2
    /* unchanged code (c1) */
  #endif
```



To detect this pattern, we start from a node with code type **code** and diff type **none** and, as for the *WrapCode* and *UnwrapCode* pattern, inspect the two paths to the root node following the **before parents** or **after parents**, respectively. If the path following the **after parents** contains at least one annotation node with diff type **add** and the path following the **before parents** contains at least

one annotation node with diff type **remove**, this pattern is matched. We also need to cover the rare case in which the presence condition changes because of multiple moved **#endif** annotations. When there are no nodes with diff type **add** or **remove** in the paths to the root node and the length of paths is equal, we compare both paths. When the paths are not equal we also match this pattern. The node c3 in the following example will thus match the *ChangePC* pattern even though no **#if** annotations were changed:

```
  #if m1
    /* unchanged code (c1) */
+ #endif
  #if m2
    /* unchanged code (c2) */
- #endif
    /* unchanged code (c3) */
+ #endif
- #endif
```



3.3.2.2 Mutual Exclusivity and Exhaustion of Atomic Patterns

We defined the atomic patterns to be mutually exclusive and provide a proof in this section. For code nodes with diff type **add**, we match the following patterns:

$$\text{matched pattern} = \begin{cases} \textit{AddWithMapping} & \text{, parent has diff type } \textbf{add} \\ \textit{AddToPC} & \text{, otherwise} \end{cases} \tag{3.2}$$

Thus, both patterns are mutually exclusive and exhaustive regarding all nodes with diff type **add**.

For nodes with diff type **remove** we match the following atomic patterns:

$$\text{matched pattern} = \begin{cases} \textit{RemWithMapping} & \text{, parent has diff type } \textbf{remove} \\ \textit{RemFromPC} & \text{, otherwise} \end{cases} \tag{3.3}$$

Both patterns are thus mutually exclusive and exhaustive regarding all nodes with diff type **remove**. All edited code nodes (i.e., nodes with code type **code** and diff type **add** or **remove**) are thus matched to exactly one atomic pattern.

Code nodes with diff type **none** match different patterns depending on the occurrences of diff type **add** and **remove** in the paths to the root node. Let α denote the path following the **after parents**, β the path following the **before parents**, $|\alpha_{add}|$ the amount of nodes with diff type **add** in α, and $|\beta_{rem}|$ the amount of nodes with diff type **remove** in β. For all cases in which at least one node in a

path was added or removed (i.e., $|\alpha_{add}| \neq 0 \vee |\beta_{rem}| \neq 0$), we get the following matches:

$$\text{matched pattern} = \begin{cases} \textit{WrapCode} & , |\alpha_{add}| > 0 \wedge |\beta_{rem}| = 0 \\ \textit{UnwrapCode} & , |\alpha_{add}| = 0 \wedge |\beta_{rem}| > 0 \\ \textit{ChangePC} & , |\alpha_{add}| > 0 \wedge |\beta_{rem}| > 0 \end{cases} \tag{3.4}$$

For all other cases in which no node was added or removed in both paths (i.e., $|\alpha_{add}| = 0 \wedge |\beta_{rem}| = 0$), we compare the lengths of both paths. Let $|\alpha|$ and $|\beta|$ denote the lengths of the respective paths:

$$\text{matched pattern} = \begin{cases} \textit{WrapCode} & , |\alpha| > |\beta| \\ \textit{UnwrapCode} & , |\alpha| < |\beta| \\ \textit{ChangePC} & , |\alpha| = |\beta| \wedge \alpha \neq \beta \\ \textit{No pattern} & , \text{otherwise} \end{cases} \tag{3.5}$$

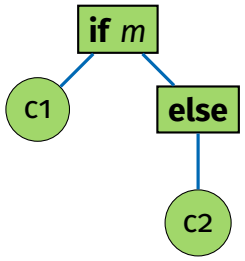
All patterns are thus mutually exclusive. The only code nodes that do not match any pattern are code nodes with diff type **none** that do not have any nodes with diff type **add** or diff type **remove** in the paths to the root node and for which both paths are equal. These code nodes are thus intended to not match any pattern as neither themselves nor their presence condition was subject to change.

3.3.2.3 Detecting Semantic Patterns

We identified 4 patterns of Stănciulescu et al. and a new pattern, the *AddIfdefElif* pattern to be semantic patterns. These patterns each consist of one or multiple atomic patterns. The semantic patterns thus overlap and are not exhaustive but describe larger changes performed by a developer. We match these patterns starting from annotation nodes in the diff tree.

AddIfdefElse:
The *AddIfdefElse* pattern describes the addition of an **`#if`**-**`#else`**-**`#endif`**-block with both branches containing added code. The pattern can look like this:

```
+ #if m
+   /* added code (c1) */
+ #else
+   /* added code (c2) */
+ #endif
```

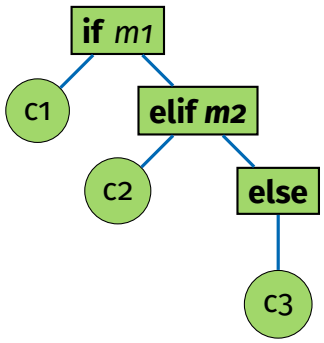


To detect this pattern, we start from a node with code type **if** and diff type **add**. This node needs to have at least two children: One node with code type **code** and diff type **add** (c1) and a node with code type **else** and diff type **add**. The else

node needs to have a child with code type **code** and diff type **add** (c2). This pattern thus contains at least two *AddWithMapping* atomic patterns. As the feature mapping of the **#else**-branch is the negation of *m*, the two atomic patterns will also have two feature mappings which are a negation of each other.

AddIfdefElif:
This is a new pattern we identified which offers a better differentiation between different annotations. This pattern describes the addition of an **#if**-**#elif**-**#endif**-block with at least one **#elif** and an optional **#else**. It could look like this:

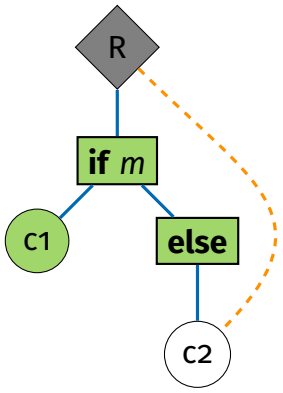
```
+ #if m1
+   /* added code (c1) */
+ #elif m2
+   /* added code (c2) */
+ #else
+   /* added code (c3) */
+ #endif
```



Similarly to the *AddIfdefElse* pattern this pattern is detected starting from a node with code type **if** and diff type **add**. This node needs to have a child with code type **code** and diff type **add** (c1 in the example) and a child with code type **elif** and diff type **add**. This **elif** child also needs to have a child which is an added code node (c2 in the example) and may recursively contain other **elif** nodes. This chain of nodes may end in a node with code type **else**. All **elif** and **else** nodes must have at least one child of code type **code** and diff type **add**. The pattern thus contains $n \geq 2$ *AddWithMapping* patterns with mutually exclusive feature mappings.

AddIfdefWrapElse:
This pattern describes the addition of an **#if**-**#else**-**#endif**-block where the **#if**-branch contains added code and the **#else**-branch contains code that existed before and is now wrapped by the annotation. An example for the pattern looks like this:

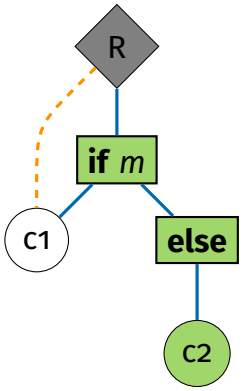
```
+ #if m
+   /* added code (c1) */
+ #else
    /* wrapped code (c2) */
+ #endif
```



Note that instead of the root node there could also be further surrounding annotations. The detection of the pattern is the same as for the *AddIfdefElse* pattern but the **else** node needs to contain at least one child with code type **code** and diff type **none**. The pattern contains two atomic patterns: An *AddWithMapping* pattern for the code in the **#if**-branch and a *WrapCode* pattern for the code in the **#else**-branch. As with the *AddIfdefElse* pattern, the feature mappings of these patterns are a negation of each other.

AddIfdefWrapThen:
This pattern describes the analog to the *AddIfdefWrapElse* pattern. Here, code is added in the **#else**-branch and wrapped in the **#if**-branch. The pattern could look like this:

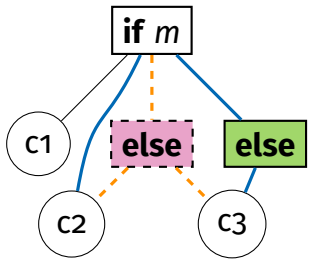
```
+ #if m
    /* wrapped code (c1) */
+ #else
+   /* added code (c2) */
+ #endif
```



Note that instead of the root node there could also be further surrounding annotations. The detection of this pattern is the same as for the *AddIfdefWrapElse* pattern but the child of the **if** node needs to be of diff type **none** and the child of the **else** node needs to be of diff type **add**. It also contains the same two atomic patterns but in different branches of the annotation block.

MoveElse:
This pattern describes moving an **#else**-statement to change the feature mapping of a block of code. It could look like this:

```
#if m
  /* code (c1) */
- #else
  /* code (c2) */
+ #else
  /* code (c3) */
#endif
```



We detect this pattern starting from a node with code type **else** and diff type **remove**. We then check if the parent of this node (**if** *m*), which can have code type **if** or **elif**, has another child with code type **else** but with diff type **add**. To capture if there was actually code between the two **else** nodes, we need to check two different options. First, the `#else` annotation could have been moved down as seen in the example. In this case, a previous child of the removed **else** node is now a child of the parent node. This is the case for c2 in the example. The contained atomic pattern is then an *UnwrapCode* pattern. Second, if the `#else` annotation was moved up, a previous child of the parent node is now the child of the added **else** node. The contained atomic pattern is then a *WrapCode* pattern. The feature mapping of the code in-between the `#else`'s is in both cases actually negated, in the example from $\neg m$ to m.

3.3.3 Step 3: Evaluation

After matching patterns, we can use the data gathered to evaluate feature trace recording. To perform our evaluation, we first have to reverse engineer the feature context for each given pattern match. By doing this we want to find out what feature contexts would have been necessary to perform the exact same changes that we detected in the commit history. We also briefly look at the number of different variants that need to be edited with feature trace recording to perform certain changes. The semantic patterns are especially relevant for this as they classify larger edits that may need to be done at once.

3.3.3.1 Determining Feature Contexts for Atomic Patterns

As feature trace recording works with an AST and DiffDetective does not create an AST from the source code files, we approximate which parts of the presence condition feature trace recording could automatically infer. We do this by just looking at the innermost surrounding annotation block and disregarding other possible surrounding feature mappings. The reverse engineering of the feature context is based on the feature contexts found for the patterns of Stănciulescu et al. in the existing evaluation of feature trace recording [BST+].

For the *AddWithMapping* and the *AddToPC* pattern the reverse-engineered feature context is just the mapping given by the parent annotation node. This is similar to the results for the *AddIfdef* and *AddNormalCode* pattern in the existing evaluation of feature trace recording. In the case of the *AddToPC* pattern

Atomic Pattern	Feature Context
AddWithMapping	m
AddToPC	m
RemWithMapping	$null$ OR $\preceq m$
RemFromPC	$\preceq m$

m is the feature mapping of the parent annotation node
$\preceq m$ means m or any formula weaker than m

Table 3.1: Atomic Patterns With Reverse-Engineered Feature Contexts

with the root node as parent, the feature mapping and reverse-engineered feature context are set to *true*. For the *RemoveMapping* pattern there are several options what the feature context could be reverse engineered to. The feature context could be omitted (i.e., set to *null*), set to the mapping m of the removed annotation node, or any formula weaker than m. The desired edit can be performed with any of these feature contexts. This is the same as with the *RemIfdef* pattern found in the existing evaluation of feature trace recording [BST+]. For the *RemFromPC* pattern the feature context can be any formula weaker than the mapping of the surrounding annotation node. In this case, there could also be no surrounding annotation, which leaves the only possible feature context to be *true*. The possible feature contexts for the patterns can also be found in Table 3.1. The other three patterns describe changes to the presence condition of existing code. Even though it would be possible to recreate these patterns using feature trace recording, these operations are an orthogonal concern and could be performed using other tools. We will thus not reverse engineer the feature context for these patterns in this thesis.

3.3.3.2 Determining Feature Contexts for Semantic Patterns

More precise information about possible feature contexts and the variants that need to be changed in larger edits can be gathered from the semantic patterns. For the evaluation of feature trace recording, we look at each pattern except for the *MoveElse* pattern as it just describes changing the feature mapping of existing code. The *AddIfdefElse* pattern was evaluated in the existing evaluation of feature trace recording [BST+]. It can only be recreated using two variants because code is added with mutually exclusive feature mappings. The feature contexts are set to the feature mapping m for the **`#if`**-branch and to $\neg m$ for the **`#else`**-branch. For the *AddIfdefElif* pattern the number of variants that need to be modified increases with the number of different branches, as all **`#elif`**-branches also have mutually exclusive feature mappings. Here, the feature contexts are also set to the feature mappings of the respective branches. The *AddIfdefWrapElse*- and *AddIfdefWrapThen* patterns were also described in the existing evaluation of feature trace recording [BST+]. These patterns can be recreated using a single feature context in a single variant. For the *AddIfdefWrapElse* pattern, the existing code needs to be removed using the feature context m so that the feature mapping changes to $\neg m$. This effectively wraps the existing code (i.e., the **`#else`**-branch). The new code can then be added using

Semantic Pattern	Feature Context	Amount of Variants
AddIfdefElse	m, $\neg m$	2
AddIfdefElif	$m_1, ..., m_n$	n
AddIfdefWrapElse	m	1
AddIfdefWrapThen	$\neg m$	1

m is the feature mapping of the **`#if`**-annotation

Table 3.2: Semantic Patterns With Reverse-Engineered Feature Contexts and Amount of Variants That Need to Be Edited

the same feature context, effectively creating the **`#if`**-branch. A similar operation needs to be performed to recreate the *AddIfdefWrapThen* pattern. With the feature context $\neg m$, the existing code needs to be removed, giving it the feature mapping m and effectively wrapping it in the **`#if`**-branch. The new code can then be added using the same feature context. The feature contexts and amount of variants that need to be edited for each pattern can be found in Table 3.2.

3.4 Summary

In this chapter, we described the main concept of this thesis, mainly revolving around the definition and detection of preprocessor-based edit patterns. The existing evaluation of feature trace recording [BST+] used edit patterns presented by Stănciulescu et al. [SBWW16]. We discovered several problems such as ambiguous descriptions and overlapping patterns which led us to refine and extend these patterns. Our refinements include a new classification for edit patterns. Atomic patterns are mutually exclusive and exhaustive regarding all changed lines of code. This is a great improvement, as it enables these patterns to be used to exactly classify all edits in the commit history of a software product line. Semantic patterns contain one or several atomic patterns and describe larger changes. They may overlap and we do not claim exhaustion for these patterns, as new semantic patterns can always be introduced. The semantic patterns enable a more in-depth analysis of the actual edits performed by developers.

With the diff tree, we introduce a novel way of representing the differences of a changed file. We provide an algorithm to generate this tree and exact definitions for matching all edit patterns. This way, both the diff tree and the matched patterns for a patch are always unambiguously defined. This makes all results of our thesis reproducible and provides clear specifications that can be used for future work on the topic.

The detection of edit patterns is the basis of our evaluation. In Chapter 4, we present DiffDetective, which is the tool we built to perform the edit pattern detection and the reverse engineering of the feature context automatically. In

Chapter 5, we present our research questions and the results of our evaluation. The research questions are based on the edit pattern detection and the reverse engineering of the feature context which were both introduced in this chapter.

4. Implementation

In this chapter, we present *DiffDetective* which is the tool we implemented to perform the evaluation of feature trace recording automatically. We describe how the tool performs the different steps that we introduced in Chapter 3. The tool is written in Java and we will provide examples for relevant parts of the source code. The complete source code can be found on GitHub[1].

4.1 Overview of DiffDetective

DiffDetective follows a pipe-and-filter structure. Starting with a git repository as input, it gathers and transforms data in several steps. The main workflow of DiffDetective is visualized in Figure 4.1. The different phases on the right side closely resemble the steps presented in Chapter 3. First, DiffDetective loads the git repository that will be analyzed. The `GitLoader` uses the `JGit` library[2] to return a `Git` object of a repository. This first step is presented in detail in Section 4.2. From the `Git` object, the `GitDiffer` gathers a list of all commits which it transforms into a list of patches and their corresponding diff trees. The result of the `GitDiffer` is a `GitDiff` object which contains a list of all patches. Each patch includes its diff tree. The `GitDiffer` implements Algorithm 1 and will be introduced in more detail in Section 4.3. In the next step, presented in Section 4.4, the `GDAnalyzer` matches patterns in the diff trees of the patches. It achieves this by checking each node in the diff trees for all of the patterns that were presented. Both atomic and semantic patterns are detected and new patterns can easily be implemented. The result of the `GDAnalyzer` is a `GDAnalysisResult` object. This object contains information about all pattern matches found in the complete `GitDiff`. The `GDEvaluator` uses this object in the last step, the evaluation, which we present in detail in Section 4.5. For each pattern match, the `GDEvaluator` reverse engineers the feature context. This information can then

[1] https://github.com/SoftVarE-Group/DiffDetective
[2] https://www.eclipse.org/jgit/

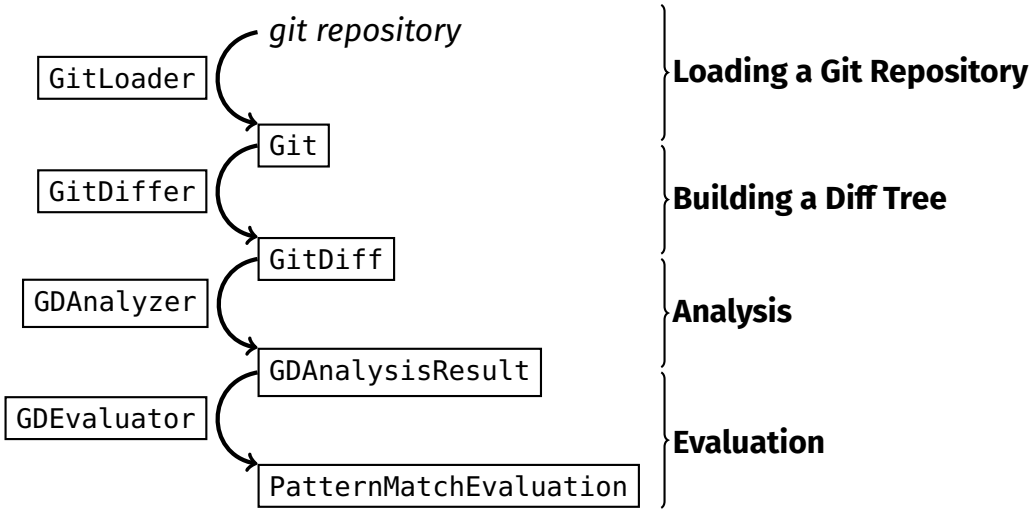


Figure 4.1: The Main Structure of DiffDetective

be printed or exported. The `GDEvaluator` also contains several methods which calculate metrics relevant to our research questions.

4.2 Loading a Git Repository

The `GitLoader` offers simple static methods for getting a `Git` object from a git repository. Usually, a git repository is cloned to a local directory. The `Git` object can then be retrieved using the `GitLoader::fromDirectory` method. As we want to be able to simply share existing git repositories, we also implemented the `GitLoader::fromZip` method, which loads a `Git` object from a zipped git repository. This method first unzips the repository using the *zip4j* library[3] and then loads the `Git` from the unzipped directory. This provides us with a simple way of sharing the version of the Marlin repository that we use for our evaluation. The last option for retrieving a `Git` object is the `GitLoader::fromRemote` method which clones a remote repository to a local directory. It then retrieves the `Git` object from the local repository.

4.3 Building a Diff Tree

The `Git` object of a git repository is the main input of the `GitDiffer`. The `GitDiffer` retrieves all commits of the repository, creates patches, and builds diff trees. The output of the `GitDiffer` is a `GitDiff` object which contains all differences found in all commits in the history of the git repository. The `GitDiffer` also receives a `DiffFilter` object which specifies which commits and which patches should be included in the final `GitDiff` object. An overview of the different options of this filter can be found in Table 4.1.

We start the creation of the `GitDiff` object in the `GitDiffer::createGitDiff` method. This method uses *JGit* to retrieve a list of commits from the given `Git`

[3] https://github.com/srikanth-lingala/zip4j

Option	Description
allowedFileExtensions	List of allowed file extensions for patches
blockedFileExtensions	List of blocked file extensions for patches
allowedChangeTypes	List of allowed change types for patches
allowedPaths	Regex of allowed file paths for patches
blockedPaths	Regex of blocked file paths for patches
allowMerge	Whether merge commits are filtered

Table 4.1: Options for the DiffFilter

object. This list of commits is then iterated. For each commit that is not filtered by the DiffFilter, a CommitDiff object is created using the GitDiffer::createCommitDiff method. This method first retrieves the complete differences of the given commit using *JGit*. For each changed file, it then calls the GitDiffer::createPatchDiff method which calls the GitDiffer::createDiffTree method.

GitDiffer::createDiffTree is the implementation of Algorithm 1. A shortened version of the implementation can be seen in Listing 4.1. The full implementation includes additional error handling when the annotations in the patch are not valid and two arguments for optimizing the diff tree: collapseMultipleCodeLines and ignoreEmptyLines. With collapseMultipleCodeLines, multiple consecutive code lines with the same diff type are put into a single node. If ignoreEmptyLines is set, all empty lines with any diff type are ignored. The implementation of the algorithm mostly follows the pseudocode definition that we presented. In lines 10-12, a new DiffNode object is created which contains the code type and diff type of the current line. At this point, the node is not yet added to the diff tree. This happens in lines 16-17 and lines 53-54 for code nodes and annotation nodes, respectively. The DiffTree object that is returned in line 58 consists of two lists that contain all code nodes and all annotation nodes.

The DiffTree is then added to the PatchDiff in GitDiffer::createPatchDiff. The class structure of the GitDiff object which is eventually returned by the GitDiffer::createGitDiff method is shown in Figure 4.2. A GitDiff object consist of n CommitDiff objects for n different commits. A CommitDiff object contains m PatchDiff objects for m different files that were changed in the commit. A PatchDiff object contains the full line-based difference of the patch in the *git diff* notation and a DiffTree object which contains the diff tree. The diff tree has DiffNode objects for each code and annotation node.

Figure 4.2: Composition of the GitDiff Class

```
1  private DiffTree createDiffTree(String fullDiff,
2                                  boolean collapseMultipleCodeLines,
3                                  boolean ignoreEmptyLines) {
4      /* ... */
5      DiffNode root = DiffNode.createRoot();
6      beforeStack.push(root);
7      afterStack.push(root);
8      for (int i = 0; i < fullDiffLines.length; i++) {
9          /* ... */
10         DiffNode newNode = DiffNode.fromLine(fullDiffLines[i],
11                                              beforeStack.peek(),
12                                              afterStack.peek());
13         /* ... */
14         if (newNode.isCode()) {
15             /* ... */
16             codeNodes.add(newNode);
17             addChildrenToParents(newNode);
18
19         } else if (newNode.isEndif()) {
20             /* ... */
21             if (!newNode.isAdd()) {
22                 /* ... */
23                 // pop the relevant stacks until an if node is popped
24                 DiffNode popped;
25                 do{
26                     popped = beforeStack.pop();
27                 }while(!popped.isIf() && !popped.isRoot());
28                 /* ... (error handling) */
29             }
30             if (!newNode.isRem()) {
31                 /* ... */
32                 // pop the relevant stacks until an if node is popped
33                 DiffNode popped;
34                 do{
35                     popped = afterStack.pop();
36                 }while(!popped.isIf() && !popped.isRoot());
37                 /* ... (error handling) */
38             }
39         } else {
40             /* ... */
41             // push the node to the relevant stacks
42             if (!newNode.isAdd()) {
43                 /* ... */
44                 beforeStack.push(newNode);
45             }
46             if (!newNode.isRem()) {
47                 /* ... */
48                 afterStack.push(newNode);
49             }
50             /* ... */
51             annotationNodes.add(newNode);
52             addChildrenToParents(newNode);
53         }
54     }
55     /* ... */
56     return new DiffTree(codeNodes, annotationNodes);
57 }
```

Listing 4.1: Implementation of the Diff Tree Algorithm

4.4 Analysis

In this step, DiffDetective detects edit patterns in the `GitDiff` object. The `GDAnalyzer` (short for `GitDiffAnalyzer`) is an abstract class used to analyze a `GitDiff` (i.e., detect patterns). It receives the `GitDiff` which will be analyzed and an array of `EditPattern` objects specifying the edit patterns that will be detected. The `GDAnalyzer::analyze` method, calls `GDAnalyzer::analyzePatch` for each `PatchDiff` in the supplied `GitDiff`. The result of the analysis is a `GDAnalysisResult` object which has a similar structure to a `GitDiff` object as it is composed of `CommitDiffAnalysisResult` objects which each consist of `PatchDiffAnalysisResult` objects. To export the results of the analysis, `GDAnalysisUtils` offers a method with which a `GDAnalysisResult` can be saved to a *CSV* file.

4.4.1 `TreeGDAnalyzer`

For the analysis on the diff tree we use the `TreeGDAnalyzer` which extends the `GDAnalyzer`. It contains two arrays of `EditPattern` objects: an array of `AtomicPattern` objects and an array of `SemanticPattern` objects. In the constructor of the `TreeGDAnalyzer` there are options for using only the atomic patterns, only the semantic patterns, or both arrays combined. The implementations of the edit patterns are presented in Section 4.4.2. The `TreeGDAnalyzer` implements the `GDAnalyzer::analyzePatch` method. First, for each `DiffNode` in the `DiffTree` with code type **code** the `AtomicPattern::getMatch` method is called. If the pattern matches, the returned `PatternMatch` object is added to a list. Second, for each annotation `DiffNode` in the `DiffTree` the `SemanticPattern::getMatch` method is called. Again, all `PatternMatch` objects are added to the list. The resulting list is then wrapped in a `PatchDiffAnalysisResult` object which the method returns.

4.4.2 `EditPattern`

All edit patterns are implemented in subclasses of the abstract `EditPattern` class which has a generic type specifying the object on which the pattern tries to match. The class mainly has two relevant methods: `EditPattern::getMatch` and `EditPattern::getFeatureContexts`. For the analysis, we will focus only on the `EditPattern::getMatch` method, which takes an object of the generic type and returns a list of `PatternMatch` objects. A `PatternMatch` object consists of the `EditPattern` that was matched, the lines in the git diff where the match is located, and an array of `Node` objects specifying the feature mappings that were found in the match. The `Node` class, which is part of *FeatureIDE* [Fea20] [MTS+17], is an implementation of a propositional formula that we use for feature mappings and feature contexts.

The abstract classes `AtomicPattern` and `SemanticPattern` both extend `EditPattern` by setting the generic type to `DiffNode`. The subclasses of them thus match patterns on `DiffNode` objects. There are 7 subclasses of `AtomicPattern`, one for each atomic pattern. The implementations of the `EditPattern::getMatches` methods follow the pattern definitions presented in Chapter 3. In Listing 4.2,

```
1  public List<PatternMatch> getMatches(DiffNode codeNode) {
2      List<PatternMatch> patternMatches = new ArrayList<>();
3
4      if (codeNode.isAdd() && codeNode.getAfterParent().isAdd()) {
5          Node fm = codeNode.getAfterParent().getAfterFeatureMapping();
6
7          PatternMatch patternMatch = new PatternMatch(this,
8                  codeNode.getFromLine(),
9                  codeNode.getToLine(), fm);
10         patternMatches.add(patternMatch);
11     }
12     return patternMatches;
13 }
```

Listing 4.2: Implementation of `EditPattern::getMatches` in the `AddWithMappingAtomicPattern` Class

we show the implementation for the `AddWithMapping` pattern. As seen in line 4, it matches on nodes with diff type **add** that have a parent with diff type **add**. The `PatternMatch` object that is created in lines 7-9 contains the pattern itself, the corresponding lines of the `DiffNode`, and the feature mapping of the parent node. For the semantic patterns, there are 5 subclasses of `SemanticPattern`. These classes are implemented similar to the implementations for the atomic pattern but they match on the annotation nodes of the diff tree.

4.5 Evaluation

We pass the `GDAnalysisResult` created by the `GDAnalyzer` to the `GDEvaluator`. This class is concerned with reverse engineering the feature context for each pattern match, calculating different metrics relevant for our research questions, and exporting this data. When the `GDEvaluator` is constructed, a `PatternMatchEvaluation` object is created for each `PatternMatch` that was found by the `GDAnalyzer`. These objects mainly contain the `PatternMatch` they are constructed from and a `FeatureContext` object. For reverse engineering the feature context, the `GDEvaluator` calls the second important method found in the `EditPattern` class: `EditPattern::getFeatureContexts`. For each implemented pattern this method returns the reverse engineered feature contexts of a pattern match. This reverse engineering follows the approach we described in Chapter 3. An example implementation of this method for the *RemWithMapping* pattern can be seen in Listing 4.3. In this case, the feature context can be either `null` (line 5) or any formula weaker than or equal to the feature mapping found in the pattern match (line 6-7).

The `GDEvaluator` contains methods for measuring the complexities of all reverse-engineered feature contexts. We define the complexity of a feature context to be the number of literals it contains. The `GDEvaluator::getFeatureContextComplexityAmounts` method does not only have a complex name but also returns the distribution of different feature context complexities. For each complexity,

```
1 public FeatureContext[] getFeatureContexts(
2                             PatternMatch patternMatch) {
3
4     return new FeatureContext[]{
5             new FeatureContext(null),
6             new FeatureContext(patternMatch.getFeatureMappings()[0],
7                                true)
8     };
9 }
```

Listing 4.3: Implementation of `EditPattern::getFeatureContexts` in the `RemWithMappingAtomicPattern` Class

it thus returns the number of pattern matches that have a reverse-engineered feature context with that complexity. Other methods return the average, minimum, and maximum feature context complexity.

We implemented methods for evaluating the number of different feature contexts found in each commit. The `GDEvaluator::getDifferentFeatureContextsPerCommitAmounts` method returns the distribution of commits requiring a specific number of different feature contexts. We calculate these values in `GDEvaluator::getDifferentFeatureContexts` which removes duplicate feature contexts with regards to equivalence and implication in a given list of feature contexts. This method is implemented using a SAT solver to determine which feature contexts are already contained in other feature context options. For instance, a feature context $F1$ and a feature context $\preceq (F1 \wedge F2)$ can be simplified to the single feature context $F1$.

The `GDEvaluator` also contains methods for exporting different results. `GDEvaluator::exportEvaluationCsv` saves all pattern matches and their reverse-engineered feature contexts to a *CSV* file. The first two columns consist of the commit hash and the name of the file that was changed which together describe a specific patch. Next, the exported file contains the name of the pattern that was matched, the feature mappings that were found, and the lines in the full git diff, where the pattern was matched. The last column contains the reverse-engineered feature contexts for the pattern match. Two other methods allow exporting the distribution of the feature context complexity and the distribution of the number of different feature contexts per commit.

4.6 Summary

In this chapter, we presented DiffDetective and its implementation. Its pipe-and-filter structure closely follows the steps we introduced in Chapter 3, but includes some optimizations to increase performance. The tool thus enables us to unambiguously detect all pattern matches in the commit history of a given software product line repository. It is easily extensible and parts of it, such as

the `GDAnalyzer` or the `GDEvaluator` can easily be replaced. With our `EditPattern` classes, new edit patterns can always be defined and integrated into the evaluation process. We provide implementations for all edit patterns described in Chapter 3 and are thus able to match a single atomic pattern on each changed line of code. With the pattern matches, DiffDetective is able to reverse engineer feature contexts and calculate metrics relevant to our research questions. We present these research questions and the complete results of our evaluation in Chapter 5.

5. Evaluation

In this chapter, we present the results of our evaluation of feature trace recording and the pattern detection. We have several research questions regarding the applicability of feature trace recording to real software projects and the occurrences of edit patterns throughout the commit history of a software product line. The subject of our evaluation is the product line *Marlin*. We gathered results to answer the research questions using DiffDetective. We also interpret and discuss the implications of our results.

First, we introduce our research questions in Section 5.1. Next, in Section 5.2, we present the study design we used for our evaluation. We present the results of this evaluation in Section 5.3. In Section 5.4, we answer the research questions using the data gathered and discuss the implications of our results.

5.1 Research Questions

For our evaluation, we answer research questions which we divide into two major questions. The first question is concerned with the results of our edit pattern detection. The second question consists of several questions concerning the applicability of feature trace recording to real software projects using different metrics.

5.1.1 RQ1: Recreating Results

> **RQ1: Can we recreate the results of the edit pattern detection on *Marlin* by Stănciulescu et al.?**

With this first research question, we aim to recreate the results presented by Stănciulescu et al. [SBWW16]. We use our implemented tool, to find edit patterns in the commit history of the *Marlin* repository. Our goal is to verify the correctness of the tool and the work of Stănciulescu et al.

5.1.2 RQ2: Evaluation of Feature Trace Recording

The following research questions are concerned with the evaluation of feature trace recording using the commit history of the *Marlin* repository.

RQ2.1: How many different feature contexts are there per commit?

For this research question, we aim to inspect the number of different feature contexts that would have been necessary in each commit to recreate the changes using feature trace recording. By inspecting how often the feature context usually needs to be switched, we can assess the effort for developers when using feature trace recording.

RQ2.2: How complex are the feature contexts?

By analyzing the complexity of the reverse-engineered feature contexts, we can get information regarding the effort for developers when using feature trace recording. We define the complexity of feature contexts to be the number of literals they contain.

RQ2.3: How often is an empty feature context (i.e., null) feasible?

In practice, it may be simpler for developers to not have to specify a feature context. As this is explicitly supported by feature trace recording, we want to gather information about edits where an empty feature context is feasible.

RQ2.4: How similar are feature contexts to the target feature mappings?

The existing evaluation of feature trace recording showed that feature contexts are often equal to or simpler than the target feature mappings. With this research question, we aim to empirically evaluate this observation.

5.2 Study Design

We use the *Marlin* repository as ground truth data for our evaluation. *Marlin* is an open-source 3d printer firmware publicly available on GitHub[1]. The project utilizes software product-line engineering using C preprocessor annotations to identify feature implementations. An advantage of using a software product line over a clone-and-own software project for this evaluation is that feature mappings are already available and do not have to be manually specified or semi-automatically recovered. Recovering feature mappings is usually error-prone and would require big manual effort. The C preprocessor is commonly used for software product-line engineering [LAL+10] so the results of the edit pattern analysis may be applicable to a wide range of software projects. As the

[1] https://github.com/MarlinFirmware/Marlin

Option	Value
allowedFileExtensions	c, cpp, h, pde
allowedChangeTypes	MODIFY
allowedPaths	Marlin.*
blockedPaths	.*arduino.*
allowMerge	false

Table 5.1: Values Used For the DiffFilter

revision of the *Marlin* repository used by Stănciulescu et al. consists of 3747 commits [SBWW16], a manual analysis of each commit is not feasible. We thus use our evaluation tool, DiffDetective, which we described in Chapter 4 to perform most of the evaluation automatically.

To answer RQ1, the recreation of the results of Stănciulescu et al., we use the same revision of the *Marlin* repository as Stănciulescu et al. We filter the commits and patches (i.e., single file changes in single commits) of the repository using the DiffFilter described in Chapter 4. The values we used for the filter can be found in Table 5.1 and follow the approach described by Stănciulescu et al. for their pattern detection. With these values, we are able to obtain the same number of patches as they did. We then analyze the filtered list of patches using the TreeGDAnalyzer. We perform two separate passes of the analysis, one detecting all atomic patterns and one detecting all semantic patterns. For RQ2, the GDEvaluator then reverse engineers the feature contexts for the relevant atomic patterns (i.e., *AddWithMapping*, *AddToPC*, *RemWithMapping*, *RemFromPC*). We do not get the feature contexts for the other atomic patterns as these patterns describe changes to the presence condition of existing code which is an orthogonal concern to feature trace recording [BST+] and thus not relevant for this evaluation. In our evaluation of the feature contexts, we also do not include the semantic patterns as we explicitly designed the atomic patterns to be mutually exclusive and exhaustive regarding all changed lines of code. Including reverse-engineered feature contexts of the semantic patterns would thus skew the data as these patterns consist of atomic patterns and may overlap. With DiffDetective, we exported all results of the evaluation which we present in Section 5.3.

Total Commits	3737
Analyzed Commits	2635
Analyzed Patches	5640
Invalid Patches	42

Table 5.2: Number of Commits and Patches Analyzed

Atomic Pattern	#patches	#matches	#lines
AddWithMapping	1363	7728	60015
AddToPC	4499	18297	62745
RemWithMapping	715	7103	53436
RemFromPC	3829	16481	57363
WrapCode	180	526	7396
UnwrapCode	68	131	534
ChangePC	2187	29706	226478
No Pattern	23	-	-

#patches: number of patches containing the pattern
#matches: total number of pattern matches
#lines: number of lines with the pattern

Table 5.3: Results of the Pattern Detection of the Atomic Patterns

5.3 Results

In Table 5.2, we show general information about the revision of the *Marlin* repository that we analyzed. As we do not evaluate merge commits, the number of analyzed commits is around 30 % smaller than the number of total commits. From these commits, DiffDetective gathers 5640 patches out of which 42 are invalid, meaning our algorithm could not generate a diff tree for them. This happens because of ill-formed preprocessor annotations such as an **`#if`** annotation without a matching **`#endif`**. We checked each of the invalid patches manually to confirm that there are no patches that were unintentionally discarded. For some of the preprocessor annotations, DiffDetective could not automatically parse the feature mapping. This mostly happened for arithmetic formulas and affected only around 0.1% of all parsed feature mappings. The feature mappings for the affected annotations were in this case automatically set to a formula containing only a single literal. Our results concerning the feature contexts may thus be simplified and biased by a small amount.

We present the results of the pattern detection for the atomic patterns in Table 5.3. The second column (#patch) shows the number of patches that include at least one pattern match of the corresponding atomic pattern. We included this number as Stănciulescu et al. only provided these values for the results of their pattern detection [SBWW16] (values from their #Multi column). We found 23 patches where no pattern could be matched. These patches all consist of either only changes in empty lines which are ignored by our implementation of Algorithm 1 as they change neither code nor feature mappings or moved **`#endif`** annotations which did not change the feature mapping of any source code. We manually checked these patches to confirm that they did not contain any unidentified patterns. The third column (#total) states the total number of pattern matches found for each pattern. This value thus equals the number of code nodes in the diff tree that DiffDetective matched to each pattern. As our

Semantic Pattern	#patches	#matches
AddIfdefElse	309	743
AddIfdefElif	72	111
AddIfdefWrapElse	64	87
AddIfdefWrapThen	47	56
MoveElse	4	6

#patches, #matches: see Table 5.3

Table 5.4: Results of the Pattern Detection of the Semantic Patterns

implementation of Algorithm 1 collapses multiple consecutive code lines of the same diff type, each code node in the diff tree may correspond to multiple code lines. In the last column (#lines), we thus present the total lines of code for each pattern not including lines with preprocessor annotations.

In Table 5.4, we show the results for the detection of the semantic patterns. The second and third columns again indicate the number of patches that contained the pattern and the number of total pattern matches, respectively. As the semantic patterns may overlap and DiffDetective is currently not able to retrieve the exact lines which are relevant for each semantic pattern, we do not include the total lines of code for the semantic patterns.

We provide different charts for results gathered by the last step of our evaluation process. For RQ2.1 we present the distribution of commits by the number of different feature contexts in Figure 5.1. The x-axis of the chart states the numbers of different feature contexts necessary to recreate a commit using feature trace recording. The y-axis shows how many commits need the respective number of feature contexts. Note that we only reverse engineered the feature context for the relevant atomic patterns (*AddWithMapping*, *AddToPC*, *RemWithMapping*, *RemFromPC*) for the reasons described in Section 5.2. We thus only include the pattern matches of these patterns in the graph. To increase readability and because at a higher number the numbers of commits are insignificantly small, we combine all commits with 15 or more different feature contexts in the last bar. The commit with the highest number of different feature contexts needs 170 different feature contexts.[2]

In Figure 5.2, we visualize the average number of changed lines of code for commits with different numbers of different feature contexts. Each mark in the chart thus describes the average number of changed lines of code (y-axis) of commits with a certain number of different feature contexts (x-axis). We gathered this data to find out whether commits with more different feature contexts are larger than commits with a low number of different feature contexts. By dividing

[2]commit: `5dabc95409b0cb011a3cc3d84772c43e39973808`

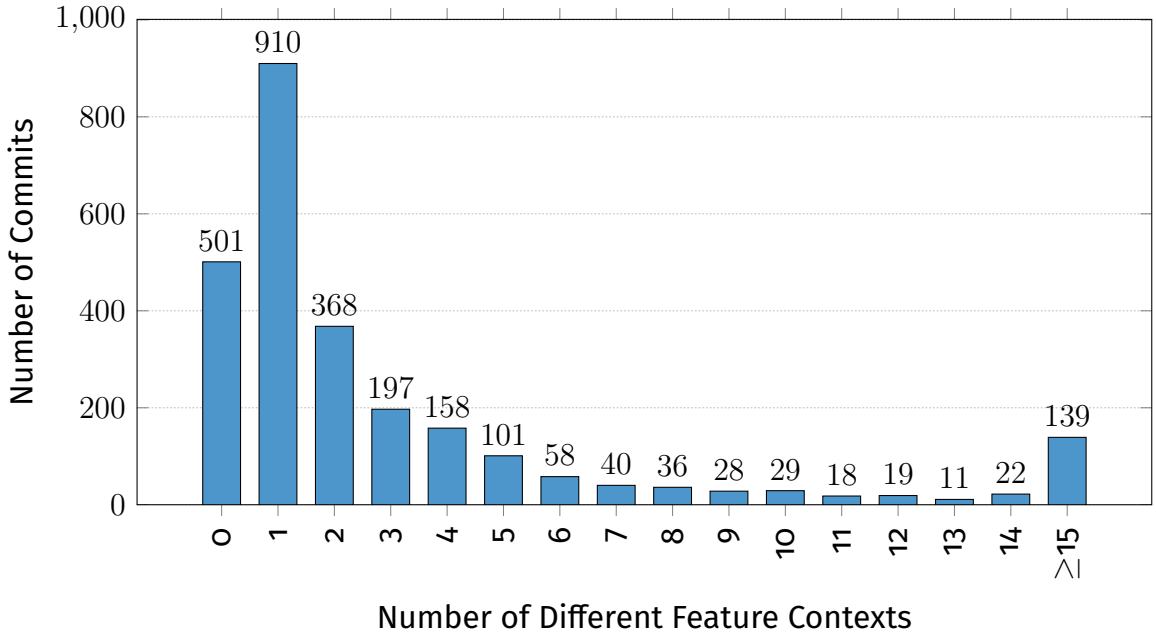


Figure 5.1: Distribution of Commits By Number of Different Feature Contexts

these values by the number of different feature contexts in each commit, we get the results visualized in Figure 5.3 which enables us to analyze the number of changed lines of code per feature context. Each mark in the chart thus describes the number of changed lines of code per feature context (y-axis) averaged for commits with a certain number of different feature contexts (x-axis). The orange dashed lines in both charts show the number of different feature contexts (15) at which we combine the values in Figure 5.1. Contrasting the total numbers of different feature contexts to the total changed lines of code we found an average of around 23 changed lines of code per feature context throughout all commits. The complete results for these graphs can be found in the Appendix in Chapter A.

In Figure 5.4 we show the distribution of atomic pattern matches by feature context complexity which we use to answer RQ2.2. The x-axis of the chart states the different feature context complexities and the y-axis shows how many pattern matches with each feature context complexity were found. Again, this data respects feature contexts reverse engineered from the relevant atomic patterns only. We combine all feature context complexities larger than or equal to 15 in the last bar. The largest complexity we found was one pattern match with a feature complexity of 53.[3] The complete results can be found in the Appendix in Chapter A.

[3]patch: `d6d6fb8930be8d0b3bd34592c915732937c6f4d9`, `Marlin/pins.h`

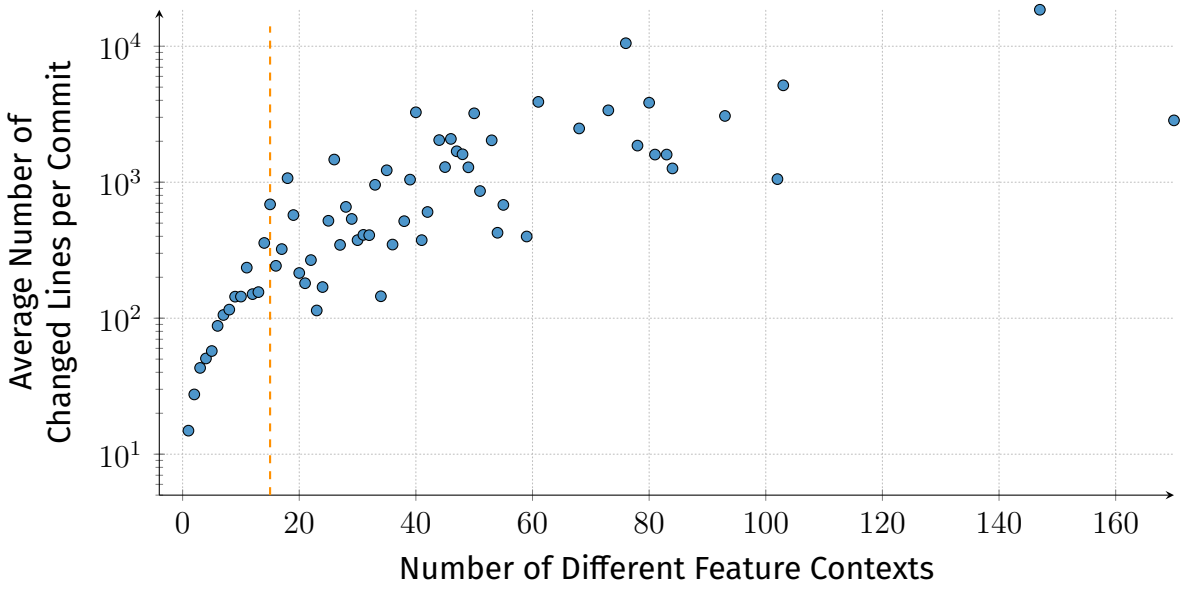


Figure 5.2: Average Number of Changed Lines per Commit For Commits With Different Numbers of Different Feature Contexts

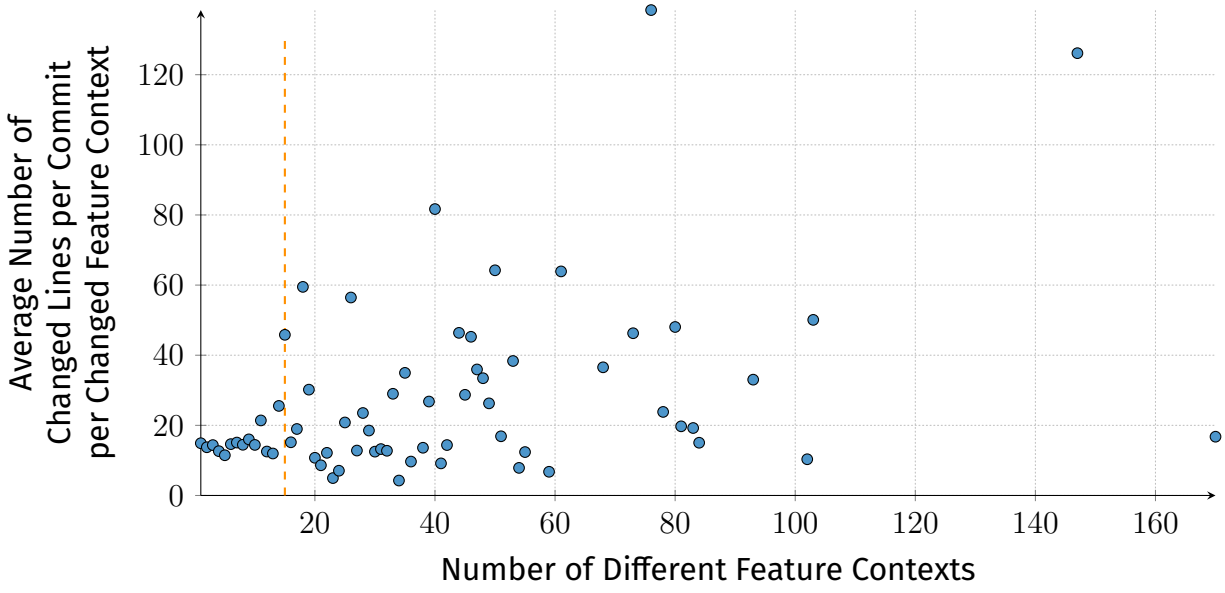


Figure 5.3: Average Number of Changed Lines per Commit Per Different Feature Context for Commits With Different Numbers of Different Feature Contexts

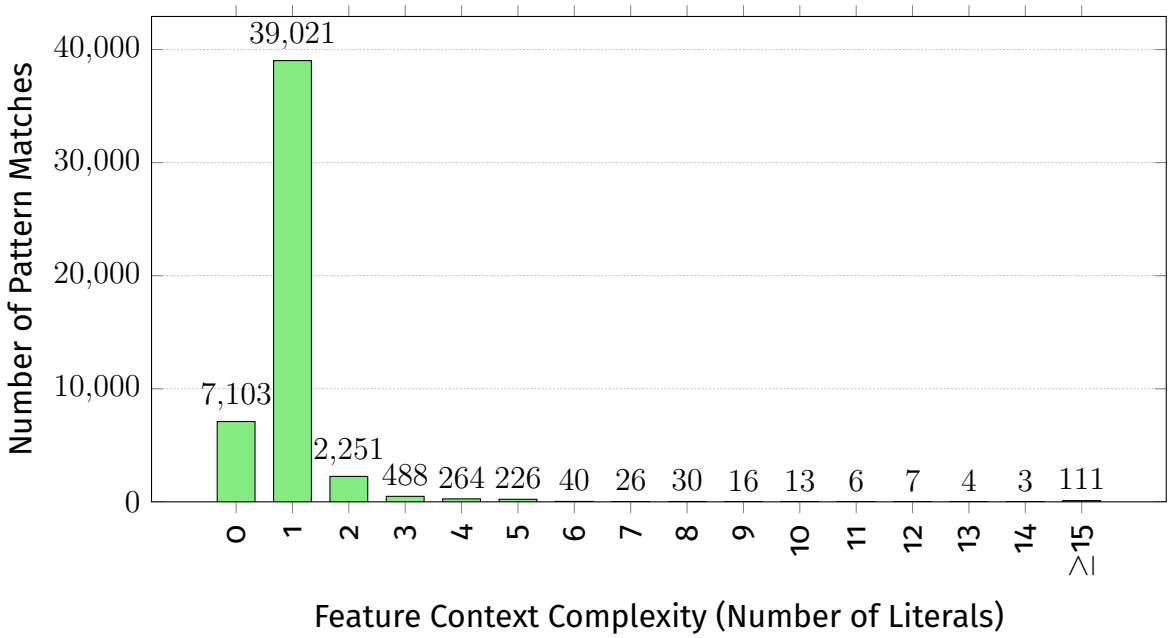


Figure 5.4: Distribution of Pattern Matches By Feature Context Complexity

5.4 Discussion

In this section, we interpret our results, answer our research questions, and briefly discuss the implications of our findings.

5.4.1 RQ1: Recreating Results

As explained in Chapter 3, we were not able to use the exact patterns of Stănciulescu et al. We refined and extended their patterns, so our results are not fully comparable. We still compare all of our patterns to the original ones and give possible explanations for differences in the number of matches. In Table 5.5, we show the numbers of patches containing our patterns and the numbers for the corresponding patterns of Stănciulescu et al. In the following, we briefly discuss the differences for each pattern.

The *AddWithMapping* pattern is best comparable to the *AddIfdef* and the *AddIfdef** pattern. From the description and the results of Stănciulescu et al. we conclude that the *AddIfdef* and *AddIfdef** pattern are mutually exclusive for each patch. We thus added up the numbers of matches for these patterns to compare them to our *AddWithMapping* pattern which matches on a slightly lesser number of patches. There are 24 patches for which Stănciulescu et al. only found either the semantic *AddIfdefElse*, *AddIfdefWrapElse*, or *AddIfdefWrapThen* pattern [SBWW16]. These patches should thus also contain at least one match for the atomic *AddWithMapping* pattern. Even if we add these patches, the difference to our result with the *AddWithMapping* pattern is very small. This difference may be caused by the invalid patches that we ignored in our analysis or changes in empty lines which we did not include. Stănciulescu et al.

Refined Pattern	#patches	Previous Pattern [SBWW16]	#patches [SBWW16]	Difference
AddWithMapping	1363	AddIfdef/AddIfdef*	1393	+ 2.20%
AddToPC	4499	AddNormalCode	4683	+ 4.09%
RemWithMapping	715	RemIfdef	534	- 25.3%
RemFromPC	3829	RemNormalCode	3932	+ 2.69%
WrapCode	180	WrapCode	77	- 57.2%
UnwrapCode	68	UnwrapCode	12	- 82.4%
ChangePC	2187	ChangePC	225	- 89.7%
AddIfdefElse	309	AddIfdefElse	271	- 12.3%
AddIfdefWrapElse	64	AddIfdefWrapElse	43	- 32.8%
AddIfdefWrapThen	47	AddIfdefWrapThen	13	- 72.3%
MoveElse	4	MoveElse	5	+ 25.0%

#patches: number of patches containing the pattern

Table 5.5: Results of Our Pattern Detection and Results of Stănciulescu et al. [SBWW16]

explicitly did not exclude whitespace changes for at least some of their patterns [SBWW16].

Our *AddToPC* pattern refines the original *AddNormalCode* pattern. Again, the difference between our result and the one of Stănciulescu et al. is relatively small. The existing difference may be caused by invalid patches or changes in empty lines.

The *RemWithMapping* pattern corresponds to the *RemIfdef* pattern. For this pattern, we found more matches than Stănciulescu et al. did which may be caused by the different definitions of the patterns. Stănciulescu et al. describe that the *RemIfdef* pattern matches edits where code is removed together with a surrounding **`#if`** block which may also contain an **`#else`** branch [SBWW16]. Our *RemWithMapping* pattern also matches on the removal of only **`#else`** branches or the removal of **`#elif`** branches. This may be the cause of the discrepancy between the matches of both patterns.

The *RemFromPC* pattern refines the *RemNormalCode* pattern of Stănciulescu et al. As with the *AddToPC* pattern, the difference between the results is relatively small and may be caused by invalid patches or changes in empty lines.

For the *WrapCode* pattern, the *UnwrapCode* pattern, and the *ChangePC* pattern, the results differ largely. This is probably caused by the different definitions of these patterns. For these patterns, Stănciulescu et al. only regarded the in-

nermost surrounding preprocessor annotation while we inspect all surrounding annotations.

Looking at the semantic patterns, our definitions for the *AddIfdefElse*, the *AddIfdefWrapElse*, and the *AddIfdefWrapThen* pattern found more matches than Stănciulescu et al. did. Stănciulescu et al. do not sufficiently explain how they handle the overlap for these patterns, so this may be a possible cause for the difference. It is also possible that we found more pattern matches because with our definition a branch of either of these patterns could additionally contain other code (e.g., an *AddIfdefElse* pattern can also contain unchanged or removed code) or further nested annotations.

For the *MoveElse* pattern we found one less match than Stănciulescu et al. did. This difference may be caused by the invalid patches, which we did not analyze, or changes in empty lines.

As expected, we were thus not able to exactly reproduce the results of Stănciulescu et al. due to several problems explained in Chapter 3. The results using our refined definitions of the patterns mostly correlate with the results of Stănciulescu et al. Differences in the results are probably caused by the invalid patches and the empty lines that we ignore or by our new definitions for patterns where the previous descriptions were ambiguous.

5.4.2 RQ2: Evaluation of Feature Trace Recording

In the following, we answer the research questions concerning feature trace recording.

RQ2.1: Different Feature Contexts per Commit

With the results in Figure 5.1, we can see that for a large portion of the commits, a small number of different feature contexts would suffice. In fact, around 54% of the analyzed commits need one or zero feature contexts and around 85% of the analyzed commits need five or fewer different feature contexts. The commits with no feature context probably consist of only edit patterns with diff type **none** (i.e., *WrapCode*, *UnwrapCode*, *ChangePC*) which are not relevant for the evaluation of feature trace recording and for which we did not reverse engineer feature contexts. For the commits with many different feature contexts, we see that generally larger numbers of feature contexts mean larger commits (i.e., more modified lines) as visualized in Figure 5.2. This explains the existence of commits with a large number of different feature contexts as these are the largest commits in the commit history. In Figure 5.3, we thus see, that a large number of different feature contexts does not necessarily indicate that edits are complex to realize with feature trace recording but that more code was edited. With a large number of feature contexts per commit, the number of changed lines per feature context does not decrease. For all values with a higher number of different feature contexts, we also see a high dispersion in both graphs which

results from the low number of commits making up this part of the data. We can thus not make any exact statements about any correlation of the number of different feature contexts in a commit and the average changed lines of code in these commits.

From our results, we can conclude that with around 23 changed lines of code per different feature context, the effort for developers may be small. As we know from the semantic patterns, some larger combinations of atomic patterns may actually be performed using a single feature context. The number of different feature contexts we found should thus only be considered an upper bound estimation.

RQ2.2: Complexity of Feature Contexts

The complexity of feature contexts is generally quite low, with 93% of the feature contexts consisting of one or zero literals and 99% of them consisting of four or fewer literals. For the larger complexities going up to a maximum complexity of 53, we found that this was caused by a large number of **`#elif`** branches. **`#elif`** branches inflate the number of literals in the reverse-engineered feature contexts as the feature mapping of each branch contains the negation of all previous branches together with a new condition. As the features in the feature mappings of these branches are often already mutually exclusive because of alternative relationships in the feature model of the product line, the feature context would in practice not actually have to include negations of all other branches.

We thus conclude that complex feature contexts are rarely needed. Our values should again be considered an upper bound estimation of the feature context complexities needed when using feature trace recording. As these values already show that 93% of feature contexts consist of one or zero literals, the effort for developers when specifying a feature context may be small.

RQ2.3: Empty Feature Context

We found 7103 matches of the *RemWithMapping* pattern which was the only pattern for which we could specify the feature context to possibly be omitted (i.e., set to `null`) without looking at complete presence conditions. In this regard, our evaluation is limited as feature trace recording works on an AST and can infer presence conditions from enclosing structures. To evaluate other cases where the feature context could be omitted, we would also need to analyze an AST which is beyond the scope of this thesis but may be part of future work on the subject. 7103 pattern matches already make up around 14% of the relevant pattern matches. The number of times where the feature context could actually be omitted in practice is thus probably much larger. The existing evaluation of feature trace recording even suggests that the feature context could be omitted for almost all patterns when the edit takes place in a scope that is already mapped to the target feature mapping [BST+].

RQ2.4: Similarity of Feature Contexts to Target Feature Mappings

In Chapter 3, we show that the feature context for the *AddWithMapping* and the *AddToPC* pattern is equal to the target feature mapping. These pattern matches make up around 52% of the relevant pattern matches. For the *RemFromPC* pattern, the feature context may also be a weaker formula and for the *RemWithMapping* pattern, the feature context can additionally be omitted. These patterns make up 33% and 14% of the relevant pattern matches, respectively.

As with the previous research question, our evaluation is limited in this regard as we do not analyze the AST of the source code. The existing evaluation of feature trace recording suggests that as with the omitted feature contexts, the feature context could also be a weaker formula than the target feature mapping when the edit takes place in a scope that is already mapped to the target feature mapping [BST+]. From our results and the pattern definitions we introduced, we can state that the feature contexts are never more complex than the target feature mappings.

5.5 Summary

In this chapter, we presented the results of our evaluation. Our first research question was concerned with reproducing the results of the edit pattern analysis of Stănciulescu et al. [SBWW16]. As we refined and extended their patterns we could not exactly recreate their results but were able to get mostly correlating numbers of pattern matches. We thus see our new exact definitions for all patterns and our atomic patterns providing mutual exclusivity and exhaustion regarding all changed lines of code as a necessary and helpful improvement.

With DiffDetective, we reverse engineered the feature contexts for the relevant pattern matches and were able to answer further research questions regarding the applicability of feature trace recording to real software projects. We found a total average of around 23 changed lines of code per feature context and showed that at least 93% of all feature contexts consist of one or zero literals. From these results, we concluded that the effort for developers when specifying a feature context may be reasonably small. We were also able to gain first insight into the feasibility of omitting feature contexts as at least 14% of the relevant pattern matches can be reproduced using an empty feature context. Our results regarding the similarity of feature contexts and target feature mappings provide a starting point for further work on the topic. All our results are reproducible and can be extended with further evaluations in the future.

6. Related Work

In this chapter, we first present related work on the topics Software Product-Line Engineering and Clone-and-Own in Section 6.1. Next, in Section 6.2, we present related work regarding feature trace recording and in Section 6.3 we look at work on edit patterns.

6.1 Software Product-Line Engineering and Clone-and-Own

Software product lines have been a part of research for many years [PBvdL05], [ABKS13], [CN01]. Various studies find many positive aspects such as an increased development efficiency, a reduced time-to-market, and higher profit margins [Kru06], [Nor08], [PBvdL05], [WL99]. An empirical analysis by Krüger and Berger [KB20] and case studies by Rubin et al. [RCC13] and Ardis et al. [ADH+00] confirm some and sometimes all of these positive aspects in practice. As Krüger and Berger also show, there is often not a clear distinction between the usage of software product lines and a clone-and-own approach in practice [KB20]. This thesis works in this spectrum as we analyze a software product line but use the data gathered to evaluate feature trace recording which works with clone-and-own software projects.

There have been many contributions on the continuum between clone-and-own and software product-line engineering and especially about the process of migrating to software product lines [AJB+14], [FMS+17], [KDO14], [KFBA09], [LC13], [WSSS16]. Different approaches range from reverse engineering of software product lines [ZHP+14] to fully functional migration tools [FMS+17]. Rubin et al. describe the steps of the migration process and present case studies on three different companies, each at a different stage of the transitioning process [RCC13]. Another case study on a successful transition to a software product line is presented by Krueger et al. [KHM06]. Laguna and Crespo present a

large study covering migrations to software product lines [LC13]. Feature trace recording can simplify a transition to a software product line because feature mappings can gradually be recorded during development using the feature context. An intermediate state, where only some feature mappings are known, is explicitly supported as the feature context can be omitted. With feature trace recording, feature mappings for a migration could be more accurate as they are recorded during development and the time needed for a migration can be reduced as recorded feature mappings do not have to be recovered [BST+].

When a full migration to a software product line is not feasible or not wanted, there are still ways in which a classic clone-and-own software project can be improved [AJB+14], [KFBA09], [GBFEKB18], [LnBC16], [LFLHE15], [SL16]. A case study by Dubinsky et al. outlines why developers might often rather use cloning in their software projects [DRB+13]. They find that developers value the simplicity and flexibility of clone-and-own approaches [DRB+13]. Fischer et al. define three informal steps for improving clone-and-own projects [FLLHE14]. First, *extraction* gets all reusable parts out of existing variants. *Composition* is the process of combining the extracted artifacts. Lastly, *completion* adds the final code segments needed to finalize the software variant. Feature trace recording could improve the extraction step by enabling developers to track feature implementations during development, when possible.

The extraction step has also been an extensive part of research on the topics *feature location* [DRGP13], *clone detection* [RBS13], and *variability mining* [KDO14]. For developers, finding implementations of features can be tedious and require a lot of time [WPXZ13] so tools to assist or automate this process are of great need. Rubin and Chechik [RC13] and Dit et al. [DRGP13] present an overview of different approaches. Some semi-automated methods only assist developers and still require a lot of user interaction [KDO14], while fully automated methods [WSSS16], [FLLHE15], [LLHE17] can often only find a fraction of the features [RC13]. Another way of addressing the feature traceability problem is to keep track of feature implementations beforehand. Ji et al. present a way of tracing features using annotations in the source code, known as embedded annotations [JBAC15]. They conclude that embedded annotations can possibly save time when recorded during development instead of recovered afterward. Their approach still requires manual annotation by the developer, which differs from feature trace recording where the feature mappings are calculated from the given feature context. The results of Ji et al. still suggest that feature trace recording could be helpful in practice. With our evaluation, we aim to empirically evaluate specifically how feature trace recording may perform regarding real software projects.

6.2 Feature Trace Recording

The existing evaluation of feature trace recording shows how common edit patterns can be reproduced but lacks definitive statements regarding the applica-

bility of feature trace recording to real software projects [BST+]. The evaluation relies on the results of Stănciulescu et al. [SBWW16] and does not include empirical data specifically gathered for the evaluation of feature trace recording. Our work is the first to present such empirical data. Additionally, we do not use the same edit patterns but refine and extend them to gain more insight into the different edits usually performed on a software product line. Our contribution also consists of extensible tool support which can be used to perform future evaluations.

There are several methods similar to feature trace recording [BST+]. An approach by Schwägerl and Westfechtel is a tool for filtered model-driven product-line engineering [SW16]. Here, only a filtered part (i.e., a partial configuration) of the whole product line is edited, whilst changes are propagated to other variants that also implement the modified features. Nguyen et al. present *JSync* which is similar to Feature Trace Recording as it also works with abstract syntax tree-based edits [NNP+12]. JSync is capable of clone detection and aids in targeted synchronization of edits with other clones of the same software. JSync tries to automatically detect changes that need to be synchronized to other clones which differs from the approach of feature trace recording, where the developer specifies which features they are working on.

While version control systems handle chronological differences between software, variation control systems manage parallel variants of software. Linsbauer et al. present a comparison of many different variation control systems [LBG17]. One of these is ECCO [FLLHE15], a tool capable of managing feature-oriented clone-and-own projects. Similarly, VTS provides support for software product lines that use the C preprocessor for defining feature implementations [SBWW16]. Feature trace recording differs from these variation control systems because it specifically allows code to not be part of any feature [BST+]. It also does not rely on a commit-based approach as the feature context can be changed at any time. Feature trace recording focuses on recording feature mappings (and thus also feature traces) and is not a variation control system. It could be used in combination with variation control systems.

Feature trace recording uses disciplined annotations on an AST meaning that features are always mapped to syntactical blocks of the source code. The implementation of a feature is thus always syntactically correct. This approach is also found in the product line tool *CIDE* [KAK08]. Different studies found several benefits of disciplined annotations as they may be less error-prone and less time consuming than the usage of undisciplined annotations [LKA11], [MRB+17], [SLSA13]. Our evaluation does not include the AST of the source code and the main subject of our evaluation, the *Marlin* repository, also utilizes undisciplined annotations. Specifically evaluating this part of feature trace recording could thus be a subject of future work on the topic.

6.3 Edit Patterns

The edit patterns that we refine and extend were initially presented by Stănciulescu et al. [SBWW16]. Stănciulescu et al. show that they can classify all patches (i.e., single file changes in single commits) in the commit history of the *Marlin* repository, but as discussed in Chapter 3 and Chapter 5 the pattern descriptions are ambiguous, sometimes overlapping and certainty on data regarding the actual amounts of pattern matches is missing. With our refined pattern definitions in Chapter 3 and the complete results of our evaluation in Chapter 5 we solve these problems. Especially the overlap of patterns is solved by the classification of atomic and semantic patterns, the former of which are mutually exclusive and exhaustive regarding all changed lines of code. Passos et al. also introduce edit patterns for preprocessor-based software product lines for their analysis of the Linux kernel [PTD+16] [PGT+13]. There are certain similarities between their and our patterns, such as their AVONMF (Add Visible Optional Non-Modular Feature) pattern corresponding to the semantic *AddIfdefElse* pattern. Their patterns differ from ours as they are also concerned with the evolution of the feature model of the Linux kernel. Their main focus lies on the evaluation of the Linux kernel and they are thus not able to make any claims about other software product lines.

7. Conclusion and Future Work

An existing evaluation of feature trace recording shows that feature trace recording is able to perform edits usually found in a software product line but does not give much insight into the applicability to real software projects [BST+]. In this thesis, we empirically evaluated feature trace recording using data gathered from the software product line *Marlin*. For this, we implemented a tool, DiffDetective, to detect edit patterns in the commit history of *Marlin*. Using the edit patterns presented by Stănciulescu et al. [SBWW16], we initially aimed to recreate their results but discovered several problems including overlapping patterns and imprecise and sometimes ambiguous descriptions. We refined and extended their patterns and one of our contributions thus consists of a new classification of preprocessor-based edit patterns with exact definitions and a method for detecting them. We present two types of patterns: atomic patterns and semantic patterns. Atomic patterns solve the problem of overlapping patterns as they are mutually exclusive and exhaustive regarding all changed lines of code. Semantic patterns describe larger edits and can give more insight into actual edits performed by developers. With DiffDetective we were able to match patterns and reverse engineer the feature context which would have been necessary to recreate the edits using feature trace recording. Our final results consist of the exact amounts for all matched patterns in the commit history of the *Marlin* repository and answers to several research questions concerning feature trace recording.

With our research questions regarding the number of different feature contexts in a single commit and the complexity of feature contexts, we aimed to evaluate the effort for developers when specifying feature contexts. We found that even though there are commits that need a large number of different feature contexts, these commits are usually just overall larger commits for which the amount of changed lines per feature context on average does not decrease. Throughout all commits, an average of 23 changed lines per feature context suggests that the feature context may not have to be switched too often. As 93% of all reverse-

engineered feature contexts consisted of one or zero literals, we concluded that the effort for developers when specifying feature contexts may be reasonably small. Our results regarding how often feature contexts can be omitted and the similarity of feature contexts and feature mappings provide first insights into these topics. As our evaluation was limited in this regard, we could not draw any exact conclusions from our results but can state that for at least 14% of all pattern matches, the feature context can be omitted.

A possible goal for further research on the topic could be to extend our evaluation using the AST and disciplined annotations. Feature contexts could be omitted more often as feature trace recording draws more knowledge from the structure of the AST. The detection of the complete presence condition even across files could increase accuracy and unveil more potential for simplifying feature contexts. With knowledge of the feature model of the evaluated product line, a further simplification of reverse-engineered feature contexts could be possible. DiffDetective can be extended to evaluate other metrics regarding feature contexts as we only looked at the complexity in terms of the number of literals they contain. Additionally, statistical relationships between edit patterns and the average complexities of the reverse-engineered feature contexts may be of interest. For our evaluation, we largely relied on the results regarding the atomic patterns as they are mutually exclusive and exhaustive regarding all changed lines of code. There may be many undiscovered semantic patterns. In general, the semantic patterns provide more insight into actual edits performed by developers but complicate an evaluation as they may overlap and are not exhaustive. We believe that our classification of the edit patterns and the results of our evaluation lay the groundwork for future (empirical) evaluations of feature trace recording.

A. Appendix

Different Feature Contexts	Amount of Commits
0	501
1	910
2	368
3	197
4	158
5	101
6	58
7	40
8	36
9	28
10	29
11	18
12	19
13	11
14	22
15	11
16	16
17	10
18	5
19	10
20	5
21	5
22	6
23	3
24	4
25	2
26	6
27	2
28	2
29	3
30	1
31	1
32	1
33	1

Different Feature Contexts	Amount of Commits
34	1
35	1
36	2
38	1
39	1
40	1
41	2
42	2
44	2
45	1
46	1
47	1
48	1
49	3
50	1
51	1
53	4
54	1
55	1
59	1
61	1
68	3
73	1
76	1
78	1
80	1
81	1
83	1
84	1
93	1
102	1
103	1
147	1
170	1

Table A.1: Amount of Commits With n Different Feature Contexts (See Figure 5.1 in Chapter 5)

Different Feature Contexts	Average Changed Lines per Commit
1	14.9
2	27.51
3	43.11
4	50.58
5	57.36
6	87.81
7	105.6
8	115.67
9	144.04
10	144.21
11	235.44
12	150.32
13	155.55
14	357.55
15	687.36
16	242.75
17	322.3
18	1070.8
19	573.4
20	215
21	180.8
22	267.5
23	114
24	169.5
25	520.5
26	1468
27	346
28	658.5
29	537
30	375
31	410
32	408
33	957
34	145
35	1224
36	348
38	517
39	1045
40	3267
41	375
42	604
44	2040.5
45	1292
46	2082
47	1689
48	1605
49	1287.3
50	3211
51	861
53	2032.75
54	425
55	681
59	399
61	3898
68	2485.3
73	3378
76	10521
78	1859
80	3844
81	1597
83	1597
84	1263
93	3072
102	1054
103	5158
147	18542
170	2849

Different Feature Contexts	Average Changed Lines per Commit per Feature Context
1	14.9
2	13.75
3	14.37
4	12.64
5	11.47
6	14.64
7	15.09
8	14.46
9	16
10	14.42
11	21.4
12	12.53
13	11.97
14	25.54
15	45.82
16	15.17
17	18.96
18	59.49
19	30.18
20	10.75
21	8.61
22	12.16
23	4.96
24	7.06
25	20.82
26	56.46
27	12.81
28	23.52
29	18.52
30	12.5
31	13.23
32	12.75
33	29
34	4.26
35	34.97
36	9.67
38	13.61
39	26.79
40	81.68
41	9.15
42	14.38
44	46.38
45	28.71
46	45.26
47	35.94
48	33.44
49	26.27
50	64.22
51	16.88
53	38.35
54	7.87
55	12.38
59	6.76
61	63.9
68	36.55
73	46.27
76	138.43
78	23.83
80	48.05
81	19.72
83	19.24
84	15.04
93	33.03
102	10.33
103	50.08
147	126.14
170	16.76

Table A.2: Values for Figure 5.2 and Figure 5.3 in Chapter 5

Complexity	Amount of Pattern Matches
0	7103
1	39021
2	2251
3	488
4	264
5	226
6	40
7	26
8	30
9	16
10	13
11	6
12	7
13	4
14	3
15	3
16	3
17	3
18	3
19	3
20	4
21	3
22	3
23	3
24	3
25	3
26	3
27	3
28	3
29	3
30	3
31	3
32	4
33	3
34	3
35	4
36	4
37	3
38	4
39	3
40	3
41	3
42	3
43	4
44	5
45	7
46	0
47	0
48	1
49	1
50	1
51	1
52	1
53	1

Table A.3: Amount of Pattern Matches With a Reverse-Engineered Feature Context Complexity of n (See Figure 5.4 in Chapter 5)

Bibliography

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013. (cited on Page 49)

[ADH+00] Mark Ardis, Nigel Daley, Daniel Hoffman, Harvey Siy, and David Weiss. Software product lines: a case study. *Software: Practice and Experience*, 30(7):825–847, 2000. (cited on Page 49)

[AJB+14] MichałAntkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Stefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. Flexible Product Line Engineering with a Virtual Platform. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 532–535. ACM, 2014. (cited on Page 1, 6, 49, and 50)

[BST+] Paul Maximilian Bittner, Alexander Schultheiß, Thomas Thüm, Timo Kehrer, Jeffrey Young, and Lukas Linsbauer. Feature Trace Recording. unpublished paper. (cited on Page 2, 3, 9, 10, 11, 24, 25, 26, 39, 47, 48, 50, 51, and 53)

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. (cited on Page 49)

[DRB+13] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An Exploratory Study of Cloning in Industrial Software Product Lines. In *Proc. Europ. Conf. on Software Maintenance and Reengineering (CSMR)*, pages 25–34. IEEE, 2013. (cited on Page 1, 6, and 50)

[DRGP13] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature Location in Source Code: A Taxonomy and Survey. *J. Software: Evolution and Process*, 25(1):53–95, 2013. (cited on Page 50)

[Fea20] Featureide repository. https://github.com/FeatureIDE/FeatureIDE, 2020. Accessed: 2020-12-22. (cited on Page 33)

[FLLHE14] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*, pages 391–400. IEEE, 2014. (cited on Page 50)

[FLLHE15] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 665–668. IEEE, 2015. (cited on Page 1, 50, and 51)

[FMS+17] Wolfram Fenske, Jens Meinicke, Sandro Schulze, Steffen Schulze, and Gunter Saake. Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line. In *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*, pages 316–326. IEEE, 2017. (cited on Page 1 and 49)

[GBFEKB18] Eddy Ghabach, Mireille Blay-Fornarino, Franjieh El Khoury, and Badih Baz. Clone-and-Own software product derivation based on developer preferences and cost estimation. In *2018 12th International Conference on Research Challenges in Information Science (RCIS)*, pages 1–6. IEEE, 2018. (cited on Page 50)

[JBAC15] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. Maintaining Feature Traceability with Embedded Annotations. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 61–70. ACM, 2015. (cited on Page 1 and 50)

[KA08] Christian Kästner and Sven Apel. Integrating compositional and annotative approaches for product line engineering. In *Proc. GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*, pages 35–40, 2008. (cited on Page 6)

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 311–320. ACM, 2008. (cited on Page 51)

[KB20] Jacob Krüger and Thorsten Berger. An empirical analysis of the costs of clone- and platform-oriented software reuse. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 432–444. ACM, 2020. (cited on Page 1 and 49)

[KDO14] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. Variability Mining: Consistent Semiautomatic Detection of Product-Line Features. *IEEE Trans. on Software Engineering (TSE)*, 40(1):67–82, 2014. (cited on Page 1, 49, and 50)

[KFBA09] Rainer Koschke, Pierre Frenzel, Andreas P. Breu, and Karsten Angstmann. Extending the Reflexion Method for Consolidating Software Variants into Product Lines. *Software Quality Journal (SQJ)*, 17(4):331–366, 2009. (cited on Page 1, 49, and 50)

[KGS+18] Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. Towards a better understanding of software features and their characteristics: a case study of marlin. In

Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), pages 105–112, 2018. (cited on Page 3)

[KHM06] Charles W Krueger, William A Hetrick, and Joseph G Moore. Making an Incremental Transition to Software Product Line Practice. *Methods & Tools*, pages 16–27, 2006. (cited on Page 49)

[Kru06] Charles W Krueger. Introduction to the emerging practice of software product line development. *Methods and Tools*, 14(3):3–15, 2006. (cited on Page 1 and 49)

[LAL+10] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 105–114. IEEE, 2010. (cited on Page 38)

[LBG17] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. A classification of variation control systems. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*, pages 49–62. ACM, 2017. (cited on Page 1 and 51)

[LC13] Miguel A. Laguna and Yania Crespo. A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring. *Science of Computer Programming (SCP)*, 78(8):1010–1034, 2013. (cited on Page 1, 49, and 50)

[LFLHE15] Lukas Linsbauer, Stefan Fischer, Roberto E. Lopez-Herrejon, and Alexander Egyed. Using Traceability for Incremental Construction and Evolution of Software Product Portfolios. In *Proc. Int'l Symposium on Software and Systems Traceability (SST)*, pages 57–60. IEEE, 2015. (cited on Page 50)

[LKA11] Jörg Liebig, Christian Kästner, and Sven 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)*, pages 191–202. ACM, 2011. (cited on Page 51)

[LLHE17] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Variability Extraction and Modeling for Product Variants. *Software and System Modeling (SoSyM)*, 16(4):1179–1199, 2017. (cited on Page 50)

[LnBC16] Raúl Lapeña, Manuel Ballarin, and Carlos Cetina. Towards Clone-and-Own Support: Locating Relevant Methods in Legacy Products. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 194–203. ACM, 2016. (cited on Page 50)

[MRB+17] 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), pages 297–307, 2017. (cited on Page 51)

[MTS+17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017. (cited on Page 33)

[NNP+12] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Clone Management for Evolving Software. *IEEE Trans. on Software Engineering (TSE)*, 38(5):1008–1026, 2012. (cited on Page 51)

[Nor08] Linda Northrop. Software product lines essentials. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2008. (cited on Page 1 and 49)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005. (cited on Page 1, 5, and 49)

[PGT+13] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wąsowski, and Paulo Borba. Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 91–100. ACM, 2013. (cited on Page 52)

[PTD+16] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. Coevolution of variability models and related software artifacts. *Empirical Software Engineering*, 21(4):1744–1793, 2016. (cited on Page 52)

[RBS13] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software Clone Detection: A Systematic Review. *J. Information and Software Technology (IST)*, 55(7):1165–1199, 2013. (cited on Page 50)

[RC13] Julia Rubin and Marsha Chechik. A Survey of Feature Location Techniques. In Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and Jorn Bettin, editors, *Domain Engineering: Product Lines, Languages, and Conceptual Models*, pages 29–58. Springer, 2013. (cited on Page 50)

[RCC13] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Managing Cloned Variants: A Framework and Experience. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 101–110. ACM, 2013. (cited on Page 1, 6, and 49)

[SBWW16] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *Proc. Int'l Conf. on Software*

Maintenance and Evolution (ICSME), pages 323–333. IEEE, 2016. (cited on Page ix, 2, 3, 9, 10, 12, 13, 26, 37, 39, 40, 44, 45, 48, 51, 52, and 53)

[SL16] Thomas Schmorleiz and Ralf Lämmel. Similarity Management of 'Cloned and Owned' Variants. In *Proc. ACM Symposium on Applied Computing (SAC)*, pages 1466–1471. ACM, 2016. (cited on Page 50)

[SLSA13] Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. Does the Discipline of Preprocessor Annotations Matter?: A Controlled Experiment. *SIGPLAN Not.*, 49(3):65–74, 2013. (cited on Page 51)

[SSW15] Stefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. Forked and Integrated Variants in an Open-Source Firmware Project. In *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*, pages 151–160. IEEE, 2015. (cited on Page 2 and 3)

[SW16] Felix Schwägerl and Bernhard Westfechtel. SuperMod: Tool Support for Collaborative Filtered Model-Driven Software Product Line Engineering. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*, page 822–827. ACM, 2016. (cited on Page 51)

[THCL] Linus Torvalds, Junio C Hamano, Scott Chacon, and Jason Long. git-diff documentation. https://git-scm.com/docs/git-diff. (cited on Page 7)

[vdZ] Erik van der Zalm. Marlin Firmware. http://marlinfw.org/. Accessed at December 02, 2019. (cited on Page 2)

[WL99] David M Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*, volume 12. Addison-Wesley Reading, 1999. (cited on Page 1, 5, and 49)

[WPXZ13] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study. *J. Software: Evolution and Process*, 25(11):1193–1224, 2013. (cited on Page 50)

[WSSS16] David Wille, Sandro Schulze, Christoph Seidl, and Ina Schaefer. Custom-Tailored Variability Mining for Block-Based Languages. In *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*, pages 271–282. IEEE, 2016. (cited on Page 1, 49, and 50)

[ZHP+14] Tewfik Ziadi, Christopher Henard, Mike Papadakis, Mikal Ziane, and Yves Le Traon. Towards a Language-Independent Approach for Reverse-Engineering of Software Product Lines. In *Proc. ACM Symposium on Applied Computing (SAC)*, pages 1064–1071. ACM, 2014. (cited on Page 49)

Declaration of Authorship

I hereby declare that this thesis is my own unaided work. All direct or indirect sources used are acknowledged as references. This paper was not previously presented to another examination board and has not been published as of yet.

Ulm, 18.04.2021

Place, Date of Submission

Signature