Technische Universität Braunschweig

Recording Feature Mappings During Evolution of Cloned Variants

Master's thesis
Sep 2018

Christopher Sontag

Institute of Software Engineering and Automotive Informatics
Prof. Dr.-Ing. Ina Schaefer
Advisor: Dr. Thomas Thüm

https://doi.org/10.24355/dbbs.084-201812191127-0

Sontag, Christopher

Recording Feature Mappings During Evolution of Cloned Variants

Master's Thesis, TU Braunschweig (DE), 2018

https://doi.org/10.24355/dbbs.084-201812191127-0

Abstract

Clone-and-Own and software product lines are often used approaches when developing variational software products. Nevertheless, both approaches have negative points which are inevitable when developing variants. While clone-and-own often is not feasible in terms of synchronizing changes into other variants, software product lines are costly when developing only a small amount of variants.

In this thesis, we tackle these problems through the creation of an annotation-based approach. Our approach uses line-based feature mappings to implement variability language independent in any textual documents. Feature mappings are recorded automatically during a source edit with as few developer interactions as possible. As automatically feature mapping is hard without knowing the intentions of developers, they select a feature context and a feature context mode which are then used by in the feature mapping calculation. With the feature mapping assigned, developers can also select patches to be synchronized to variants which also implement the calculated feature mapping. In total, our approach mixes elements from clone-and-own and software product lines and enables developer to work as efficient as possible with an acceptable cost-benefit ratio.

Eidesstattliche Erklärung

Ich versichere eidesstattlich, dass ich die vorliegende Masterarbeit *“Recording Feature Mappings During Evolution of Cloned Variants”* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Braunschweig, den 1. September 2018

Acknowledgements

I would like to thank Prof. Dr.-Ing. Ina Schaefer for the opportunity to write my thesis in this field of computer science. I also want to thank my supervisor Dr. Thomas Thüm for his overall support for this thesis and the productive discussions. He always had time to discuss questions and problems and gave me many advices and comments regarding scientific writing.

Also, I want to thank my friends for all their support during the master program and as proof-readers for this thesis. Last but not least I would like to thank my loved ones, who always have supported me throughout study and whenever I faced problems. Nothing can make up for this.

Contents

https://doi.org/10.24355/dbbs.084-201812191127-0

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 Motivation

In history, software product lines were not the standard process to go when developing software projects. Software was an individual piece of code which was developed only for specific customer. In later years as software evolved in complexity and diversity, individual software development got more expensive and challenging and standardization was needed to overcome these issues. As standards do not fulfill individual needs, software product lines gather more and more interest since the 1990s. Reusable artifacts deliver a basic variant for a domain which is then customized to the requirements of individual customers [3]. In literature and research, these customizations are separated in seven levels, the often used clone-and-own approach on level L0 (s. section 2.2) and a full software product line on level L6 (s. section 2.1) [2]. Both extremes are not the best possible solution for developing variability as clone-and-own (L0) has the negative point of propagation of changes in a multiple variant-development scenario. Additionally, an enhanced pure clone-and-own (L1) where version control systems are used to keep track about the changes, lack support for context selection during the development. On the other extreme (L5 and L6), the cost-benefit ratio is often worse as domain engineering and migration needs a high effort from developers. While L5 has an integrated platform for developing but still allows clone-and-own, level 6 has a fully integrated platform where only features are developed and no projects. Accordingly, software product lines are often not practicable for smaller numbers of variants.

Our approach aims to improve the gap between levels L1 and L5, but actually does not fit in the proposed seven levels. We want to improve clone-and-own development (L0-L1) through tool support for the application engineering. Therewith developers get the advantages of feature-to-artifact mappings and configuration management, but do not need a product-line architecture or reusable assets. Thus, the efforts can be kept low as level L5 and L6 necessaries are not needed and the efficiency of the levels L0 and L1 is maintained. While features in product lines are always in sync across all variants, we want to supply a n-way synchronization which synchronizes artifact changes across variants when needed. For this, we want to record changes of artifacts and assign individual features to them. Subsequently, these changes can then be synchronized into other variants.

https://doi.org/10.24355/dbbs.084-201812191127-0

1.2 Tasks & Objectives

Objectives

Automated recordings of changes and without user interactions, their mappings to features in form of feature contexts could improve the usability and practicability of methods which attempt to enhance clone-and-own with software product lines. Despite that, automated recordings of mappings leads to some problems like missing knowledge for mapping interactions as well as synchronization issues. Mapping and synchronizing feature contexts to their artifacts can sometimes open the question which feature context should be applied (e.g. the surroundings are already mapped, the line is already mapped, or the line is going to be removed). In these cases, propositional formulas could enhance the feature context so that changes can automatically be mapped. This thesis should create an approach for solving these issues in an conceptional framework manner with the possibilities to transfer the approach to different research ideas easily.

Tasks

1. Create a conceptional framework approach for feature context mappings. When artifacts are added, changed, or removed, the approach should provide a usable and practical way to conclude the right feature context with as lightweight user interactions as possible. Special focus should lay on interactions between feature contexts as they occur more and more during the evolution of variants and often need additional information which are mostly not provided when using automated feature context mapping.

2. Create a conceptional framework approach for synchronization of artifacts with feature context mappings. Mappings which add, change, or remove artifacts can conflict with mappings in target variants and should be resolved with usability and practicability in mind.

3. A prototypical implementation of the approaches in VariantSync [51]. To support the approaches during recording and synchronization, the line-based algorithm should be enhanced with a component that resolves interactions between feature context mappings. The implementation should also be replaceable like the rest.

4. Evaluation of the approaches based on, for example, one of the following options:
   - A small self-made software product line which is similar to an industrial product line and uses VariantSync [51] from a certain point onwards.
   - An existing product line for which the history is available. This includes both existing clones and also product lines created with different implementation techniques.

- An experiment with students or an industrial partner which uses VariantSync [51] to develop a product line variant.

In all evaluation methods, the different cases of interactions between feature contexts as well as the synchronizations between variants should be collected and evaluated in the aspects of overhead to a classical development and their practicability.

1.3 Structure of the Thesis

The thesis is split into a background, a concept, an implementation, and an evaluation chapter. In Chapter 2, we introduce the terms and necessary background informations for following chapters. We introduce our running example, the Marlin firmware [44], basic knowledge about software product-lines and also about clone-and-own. Our concept for feature mappings and synchronization is then presented in Chapter 3, where we also define feature contexts as central element of the thesis. We also show extensions, which could improve the feature mapping through more information. The implementation in VariantSync [51] is then shown in Chapter 4. We show the current work-flow and how we changed it to support our concept. In Chapter 5, we evaluate the concept under Marlin firmware [44] pull-requests before we give some insight about related work in Chapter 6 and come to a conclusion in Chapter 7. The thesis ends up with a view into future work that could be made to improve the concept in Chapter 8.

2 Background

This chapter introduces the basic terminologies which are required in this thesis. Section 2.1 presents software product lines as a variant development and management technique. Section 2.2 explains the clone-and-own approach as another way to develop and manage variants with different artifact sets and the differences to a software product line.

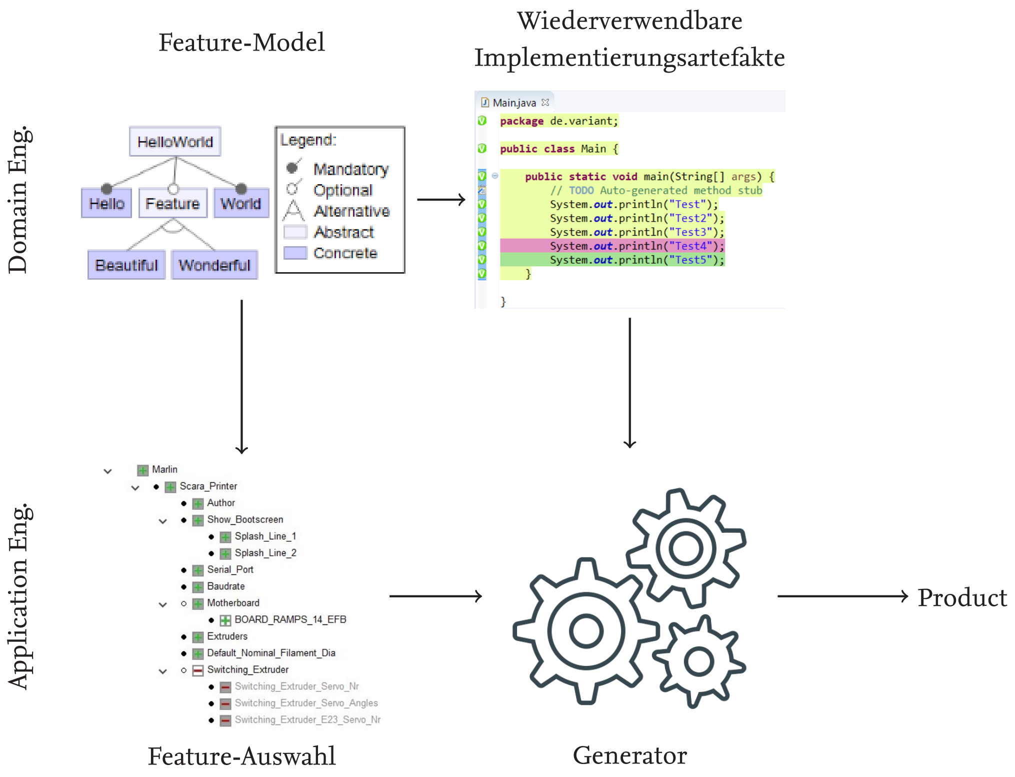
2.1 Software Product Lines

Software product-line engineering is a software development paradigm which helps to develop software of a specific domain, using software platforms and mass customization [52]. In this context, a software product line is the set of products that can be build on top of a common set of core assets and has a set of features which describes commonalities and variability points [15].

The process of software product-line engineering consists of domain engineering and application engineering as shown in Figure 2.1 [3, 52, 69]. Domain engineering is the process of analyzing the commonalities and differences of variants. As the results of this assessment, depending on the knowledge of the person performing the work, domain experts are often used for this step [26]. On top of these results, developers start implementing the domain in reusable artifacts. The development itself consists of requirement analysis, domain design, domain realization, and domain testing. At the beginning, requirement analysis evaluates and documents the feature set of the software product line. In the subsequent domain design, the reference architecture is defined and leads to detailed design and the implementation of artifacts in the domain realization, where each feature is modularly implemented for later composition. The tests in the domain testing process conclude a cycle of the domain engineering.

The subsequent process of developing a specific variant for a customer is called application engineering and uses artifacts from domain engineering. It also consists of a requirement, a design, a realization, and a testing process. The difference is the context which lies on a single variant instead the complete software product line and results in a generated variant for each customer. Through this aspect, the requirement analysis creates a set of selected features. Afterwards, the reference design is instantiated with these requirements and implemented with the reusable artifacts of the domain engineering output. The first development iteration ends with testing and delivering the single variant to the specific customer.

https://doi.org/10.24355/dbbs.084-201812191127-0



```
package de.variant;

public class Main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Test");
        System.out.println("Test2");
        System.out.println("Test3");
        System.out.println("Test4");
        System.out.println("Test5");
    }

}
```

```
Marlin
    Scara_Printer
        Author
        Show_Bootscreen
            Splash_Line_1
            Splash_Line_2
        Serial_Port
        Baudrate
        Motherboard
            BOARD_RAMPS_14_EFB
        Extruders
        Default_Nominal_Filament_Dia
        Switching_Extruder
            Switching_Extruder_Servo_Nr
            Switching_Extruder_Servo_Angles
            Switching_Extruder_E23_Servo_Nr
```

Figure 2.1: Domain and Application Engineering Process [52, 61]

2.1.1 Running Example: Marlin Firmware

As a running example in this section, we take the Marlin firmware which is a universal 3D printer firmware with many features like LCD support and dual extruder settings [44]. All examples are extracted from the bugfix-2.0.x branch. Marlin has over 5600 forks and is one of the most forked projects on Github [24]. The complete firmware has around 450 features which can be configured through preprocessor statements in configuration files. Marlin supports 59 3D printers out of the box with configuration files, which represent the most standard printers. For other printers, developers must edit the normal or the advanced configuration file.

2.1.2 Adoption Strategies

As software product lines are often not created from scratch, literature, as mentioned earlier, separates the transition for variable software in seven levels from the classical clone-and-own (L0) to software product lines (L6) with a fully integrated platform [2]. Each level gains therefore functionality which is present on the next level as well. Level L1 adds history while Level L2 adds features to the variant development and Level L3 configurations. After this, Level L4 enhance the development process with feature models before Level L5 starts with an integrated platform but still allows clone-and-own.

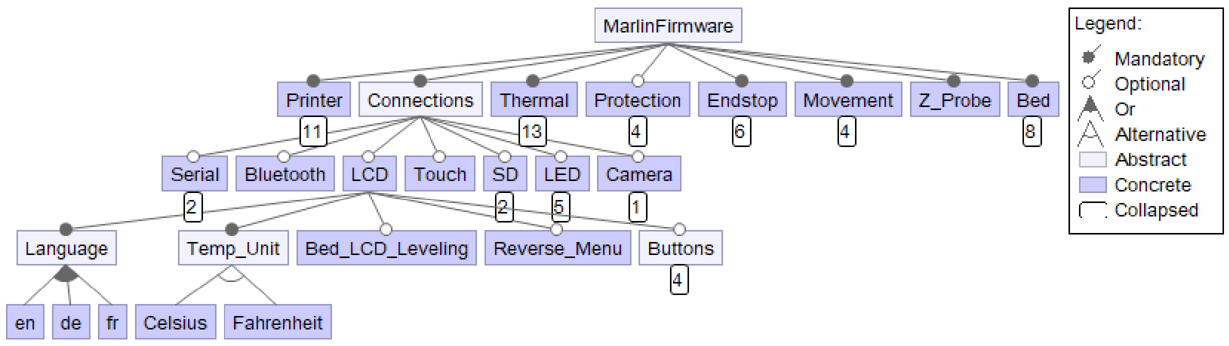
As each level must be paced, classical software development variants are often migrated to a software product line with one of three different adoption strategies which exist in literature [3, 34]:

- The proactive approach builds a product line from scratch using domain engineering. Domain experts create a domain model and developers implement all features before the first product is build. The strategy is often seen as an academic approach which has to be changed in some way to use it in the industry as it leads to high initial costs [9, 17].
- The reactive approach creates an initial software product line which is incrementally extended when new requirements rising up. It is the most agile method out of the three strategies and describes the typical maintenance phase in a software product-line life-cycle.
- The extractive approach starts with existing variants and refactors them into a software product line. The existing products can be sliced into a common set of core assets and a set of commonalities and differences. This approach is the most flexible, it has lower initially costs than the proactive approach, and existing products can be build with the old development approach until the adoption is finished. Therefore, the extractive strategy is the typical case in the industry [9, 17].

2.1.3 Feature Model

As the variability is the critical step in a software product line, a common approach besides decision models [58], orthogonal variability models [52], and use-case models [13] is to model variability in feature models [3, 19, 26, 27, 27, 60]. Feature models consist of features and relations between them. A feature describes an element in the domain and is either mandatory, optional, or a member of an alternative- or or-group. Furthermore, each feature can be abstract and not assigned to any code [3]. Between features, logical expressions can be used to express relations and dependencies. On sets of features and feature models, many conclusions can be conducted through boolean satisfiability problem (SAT) solvers [3, 7]. For example, a SAT solver can be used to check the consistency of a feature model, which means, that it has at least one feature set which generates a valid variant [3, 46]. It can also be used to check whether there are dead features or false-optional features [46, 55, 65, 66]. Features are dead if they do not belong to any configurable variant and false-optional if they are included in every variant, but are marked as optional in the feature model.

A feature model is mostly presented as a feature diagram which is typically organized in a tree structure with cross-tree constraints as shown in Figure 2.2 [3, 60]. A child feature f with parent feature p can therefore only be selected when p is also selected. In this example, the feature *Bed_LCD_Leveling* is a child of feature *LCD* and can only be selected when *LCD* is selected as the parent feature.



Touch ⇒ LCD

Auto ⇒ Bed_LCD_Leveling

Runaway ∨ Runout ⇒ Disable_Inactive

Power_Supply_Control ⇒ LCD ∨ LED

Multiple ⇒ Disable_Inactive ∧ Preheat

LCD ∧ ¬(Serial ∨ Bluetooth ∨ Touch ∨ SD) ⇒ Buttons

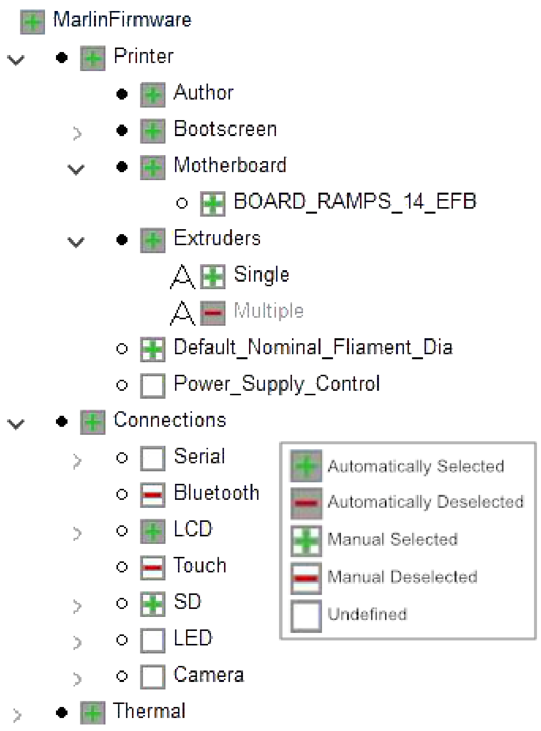
Figure 2.2: A feature model for the 3D printer firmware Marlin [44]

2.1.4 Configuration

A configuration refers to a set of selected features which should be a valid subset of the feature model and is created during the configuration process [37,50,70]. A feature in a configuration can be either selected, deselected, or undefined [50] and should match the requirements to an actual variant as shown in Figure 2.3. In this example, feature *BOARD_RAMPS_14_EFB* is explicitly selected for the variant, while the feature *Extruder* represents a mandatory selection which means a selection in consequence of a cross-tree constraint of the feature model or a selected child feature. Explicitly deselected are the features *Power_Supply_Control* and *Bluetooth* while the feature *Serial* is undefined. Whether a configuration is valid or not can be determined through satisfiability problem solving. The SAT solver proofs validity if a configuration can satisfy all dependencies of the feature model and is therefore a valid variant of the domain.

2.1.5 Feature Mapping

Generating variants or exchanging codes between them are key features for many approaches. Therefore they use feature mappings which associate features to artifacts [21,25,28]. An old but often used mapping implementation is a preprocessor, which is shown in Listing 2.1. The listing shows the implementation of the *power_on* function which controls the internal power distribution. If the feature *HAS_TRINAMIC* is defined, push is called for each stepper drivers before using the stepper again. Furthermore, each conditional preprocessor directive is defined by a presence condition which also allows complex conditions [30].



```
🟩 MarlinFirmware
  ● 🟩 Printer
      ● 🟩 Author
      ● 🟩 Bootscreen
      ● 🟩 Motherboard
          ○ 🟩 BOARD_RAMPS_14_EFB
      ● 🟩 Extruders
          🟩 Single
          🟥 Multiple
      ○ 🟩 Default_Nominal_Fliament_Dia
      ○ ☐ Power_Supply_Control
  ● 🟩 Connections
      ○ ☐ Serial
      ○ 🟥 Bluetooth
      ○ 🟩 LCD
      ○ 🟥 Touch
      ○ 🟩 SD
      ○ ☐ LED
      ○ ☐ Camera
  ● 🟩 Thermal
```

🟩	Automatically Selected
🟥	Automatically Deselected
🟩	Manual Selected
🟥	Manual Deselected
☐	Undefined

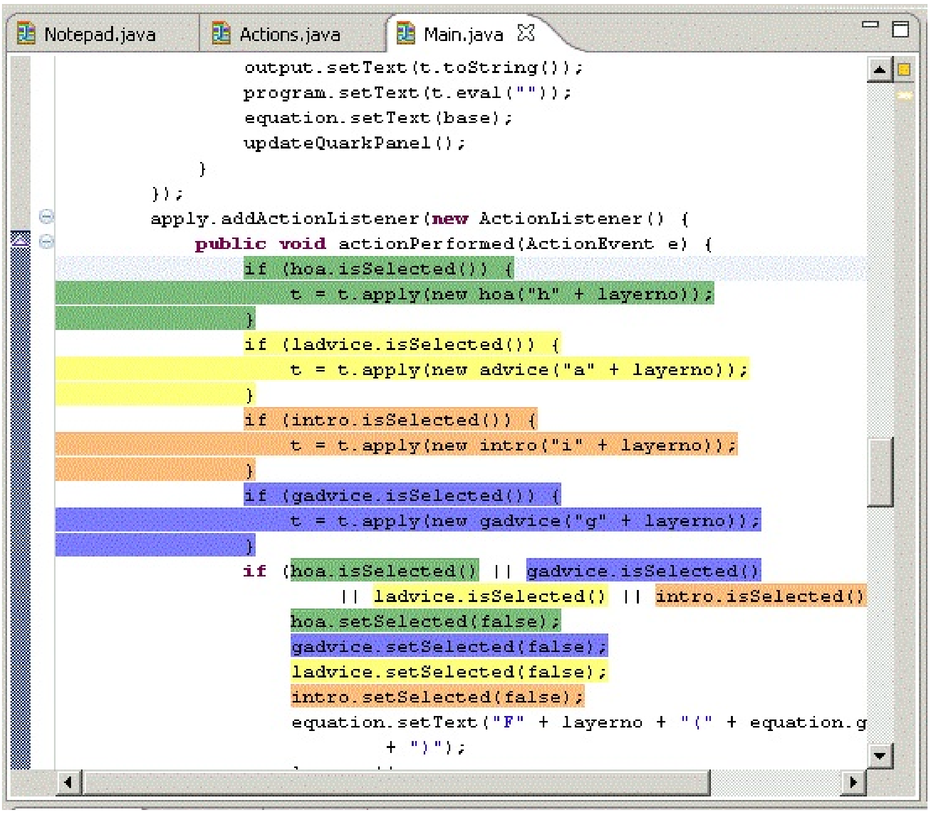
Figure 2.3: Valid configuration for the 3D printer firmware Marlin firmware [44]. It supports a LCD panel and is a single extruder printer, but misses power supply control and bluetooth.

```cpp
void Power::power_on() {
  lastPowerOn = millis();
  if (!powersupply_on) {
    PSU_PIN_ON();

    #if HAS_TRINAMIC
      delay(100); // Wait for power to settle
      restore_stepper_drivers();
    #endif
  }
}
```

Listing 2.1: Example preprocessor code for the feature *HAS_TRINAMIC. Marlin/ultralcd.cpp* [44]

While variability modeling in the domain engineering is a problem space solution, feature mappings are a solution space approach. Therewith, they enable easy identification of features in artifacts. Figure 2.4 shows CIDE [35] which enables developers to create mappings by colorizing their code. The features are displayed with a background color and developers can identify features faster



```
                output.setText(t.toString());
                program.setText(t.eval(""));
                equation.setText(base);
                updateQuarkPanel();
            }
        });
        apply.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if (hoa.isSelected()) {
                    t = t.apply(new hoa("h" + layerno));
                }
                if (ladvice.isSelected()) {
                    t = t.apply(new advice("a" + layerno));
                }
                if (intro.isSelected()) {
                    t = t.apply(new intro("i" + layerno));
                }
                if (gadvice.isSelected()) {
                    t = t.apply(new gadvice("g" + layerno));
                }
                if (hoa.isSelected() || gadvice.isSelected()
                        || ladvice.isSelected() || intro.isSelected()
                    hoa.setSelected(false);
                    gadvice.setSelected(false);
                    ladvice.setSelected(false);
                    intro.setSelected(false);
                    equation.setText("F" + layerno + "(" + equation.g
                            + ")");
```

Figure 2.4: Virtual separation of concerns in CIDE [35]

because features are hidden in the source code as color [28]. Hereby, CIDE realizes the virtual separation of concerns which means that annotation-based techniques are enhanced with tool support to emulate modularity [30]. With this, source code readability can be improved as no feature assignment is needed in the source code. The approach is also language independent as it is not based on any syntax. Nevertheless, some approaches also use syntax to enhance their process. In this context, ASTs can be used to receive an abstracted view on the parse tree, which represents the structure of the complete document [29]. CIDE also uses ASTs to enhance their feature mapping as they can determine the elements that should be assigned in the process. On the other hand, ASTs come with the price that only documents are supported which can be analyzed by a parser.

Generally, feature mapping can be achieved in both, a manual and an automated way [25]. In a manual mapping process, the mapping is completely done by the developers through assigning a feature to an artifact after implementing it. Instead, the automated mapping process can minimize the effort for developers as they e.g. analyze all artifacts and generate the mappings by extraction [21, 28]. The manual mapping leads to a n-to-n relationship where n features can be assigned to n artifacts. As there is no difference to the mapping itself when using the automated feature mapping process, the same n-to-n relationship is applicable [25].

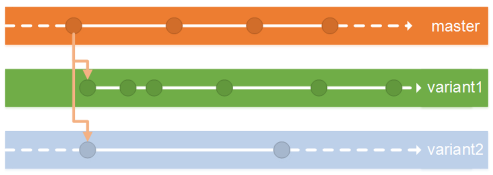


Figure 2.5: Clone-and-own workflow with git [61]

2.2 Clone-and-Own

The clone-and-own process is often used in classical software development instead of software product lines because it has low usage requirements [20]. A new variant can be build by simply cloning an existing variant and make changes to it [17, 20]. It usually consists of an extraction, a composition, and a completion phase [22]. In the extraction phase, the reusable artifacts are located in the existing variant and then composed to a new variant through the extracted artifacts. In the last step, the modification takes place so that all requirements are fulfilled. But there are also some negative factors when developing with clone-and-own. The code reuse is low as code is always cloned whether it will change or not. Also, maintaining multiple variants can be hard and code scattering is a problem depending on the number of variants involved. Artifact interactions are problematically as well, because they are leading to unnecessary code in the new variant [20, 22].

Clone-and-own can be implemented in many different technologies. One technology are version control systems (level L1, s. Section 2.1.2) which are used to create code clones using branching and forking as they are often already available for the developers [42]. F igure 2.5 shows how variants can be cloned from a master source so that changes can then be done by committing to the individual repository without changing the original source. An often used strategy for developing variants in version control systems are also feature branches where each variability point is implemented in its own code clone [42]. Nevertheless, version control systems are only level L1 because they do not support feature traceability and variant management as known in software product lines, but have a history of changes [2,20]. A solution for some problems mentioned above could be computer-aided clone-and-own [36], variation control systems [41] and other similar approaches like ECCO [21] where new variants can be extracted from existing ones based on additional information. Still all clone-and-own approaches have common problems as code scattering as well as the risk of double implementations of the same functionality is not reduced [8]. Despite that, different approaches could improve code readability as they provide tool support for this. They are usually sorted in the levels L1 to L5 as some generate even complete variants from existing ones in the application engineering process.

3 Automated Recording Feature Mappings in Source Code

In this chapter, we want to introduce our concept to improve the development of a small number of variants. As current research focuses on approaches which regain feature mappings after the edit of source code, we aim to improve feature mapping through an automatism which records the feature mapping in the moment the edit was created. The approach should automatically record feature mappings while making additions (s. Section 3.1.2), changes (s. Section 3.1.4), or deletions (s. Section 3.1.3) to variant artifacts with the best possible feature mapping. This should minimize the number of developer interactions in the mapping process which will be realized through feature contexts (s. Section 3.1.1) and feature context modes (s. Section 3.1.5). Both give the developers tools to control the calculation of a feature mapping when an edit is made. While feature contexts are presence conditions which indicate the feature or feature interaction the developer is working on, feature context modes control the influence of current feature mappings in the calculation.

With these two core elements, we want to show a line-based approach first as it is widely available for all types of artifacts, such as source code without the need of syntax or the programming language. With this, developers can create feature mappings for whole lines in a document and only need to specify the feature context and the feature context mode under which the current edit is made. The result is a feature mapped line where the feature mapping is influenced by the developer.

Nevertheless, the evaluation could show that a line-based mapping is not enough for complex variant development. Therefore, we want to introduce and later evaluate two possible improvements which can enhance the line-based approach. In Section 3.2.1, we discuss the enhancement of the line-based approach with AST‘s. Abstract syntax trees provide a structured view onto artifacts and can improve the feature mapping automatism when using structural dependencies. With this approach, we can calculate an influence probability of lines around the edit. We can also annotate different parts of an if-clause with this extension. Another improvement is introduced in Section 3.2.2. Feature models could provide informations such as features and their dependencies.

3.1 Line-Based Recording

Evolving a small amount of variants without good variant management can be a pain as propagating artifact edits to all variants is bound to a high effort and software product line migration costs are high for a small number of variants [3, 9, 17, 34]. Our concept aims to improve the situation through filling the gap between clone-and-own and software product-line development by recording feature mappings automatically while editing a variant. As calculating a new feature mapping without taking the developer's intentions into account does not work well for line-based changes, we want to make the missing information explicit using feature context and feature context modes. As explained above, feature contexts are presence conditions which describe features or feature interactions that could be implemented in every variant (s. Section 3.1.1). Also, feature context modes are boolean calculation functions which influence the calculation of new feature mappings by defining the handling of already existing feature mappings (s. Section 3.1.5). With these information made explicit by the developers, our concept can automatically calculate an adequate feature mapping with developer intentions in mind for each edit made by developers.

Besides the automation of feature mappings, we also automate the synchronization process between variants in our concept. We realized a feature-mapping-based synchronization process that takes over changes from one variant to another based on their configurations (s. Section 2.1.4), which should also minimize the number of errors while applying edits into other variants.

The complete recording and synchronization process can be split up into three phases, the intention, the editing, and the synchronization phase. Before developer start editing a line-based document, they need to take action in the intention phase which also should be the only interaction needed by them. This includes the selection of a feature context that represents the feature or feature interaction the developer will work on. It also needs a feature context mode, which controls the current feature mapping influences in the calculation of new feature mappings. With both settings, developers can regulate the semi-automated feature mapping calculation and do not need to take action until one of the intentions changes.

After the intention phase has finished, developers can start with the editing process and edit documents in variants. As we do not want to require any further developer interaction for the feature mapping calculation, we only want to relay on information that are available before, during and after the edit. An information we can determine are the lines that have changed. For this, we use a diff algorithm that takes the old and the new version of a document and calculates the edit. Often, diff algorithms differ in their edit kinds as some algorithms only support adding and removing, and not changing which we also want to use for the evaluation later. Therefore, we use Myers diff algorithm [47, 48] which also supports the change edit kind. Based on these kinds of edits, we can also create three basic operations for the feature mapping calculations. While *Add* is called when lines are added to a file, *Remove* handles feature mappings when the diff algorithm determines that lines are removed. Along with these cases, the *Change* operation calculates the feature mapping, when already existing lines change.

The last phase for an improved development process for a small number of variants is the n-way synchronization step. This is an important step as synchronizable variants for a specific edit are often only known by the developer of the edit. This leads to serious issues when multiple persons develop variants and other developers do not know about the configurations of all variants. Also, new variants can not be synchronized when the developer does not know the edits that need to be taken over to the new variant. Another problem is the feature traceability when taking edits over manually as it implies a huge effort. To solve these issues, we improve the synchronization process with our recorded feature mapping. Each edit has a list of possible synchronizable variants, which are variants that satisfy the equation

$$SAT(FM \wedge \bigwedge_{f \in C_V} f \wedge \bigwedge_{f' \in F \setminus C_V} \neg f'). \tag{3.1}$$

This means that the feature mapping corresponds with all selected and not selected features of the configuration of a variant. If the equation is satisfied, the changes can be patched either automatically or manually at any time of the development process. Automatically means in this context that no adjustments are needed to patch the edit into the other variant, while a manual patching process requires the knowledge of developers. Whether a patch can be synchronized automatically or manually also depends on the synchronization history of the variant as edits are always a successor of another edit. When an edit of a earlier time stamp is synchronized after a later one, the earlier edit can be selected to be patched manually when the same region of lines are touched. While editing the patch, we also calculate a new mapping for the patched lines with the different operations mentioned above. With this, we improve also the feature traceability in the synchronized variant when synchronizing the edit.

3.1.1 Feature Context

Feature mappings are as good as a developer specifies them. This causes a lot of problems as a automated approach can not react to the developers intention. Therefore, we introduce feature context as a way for the developer to express their intentions. With feature contexts, developers explicitly select a presence condition as working feature intention. An example is the feature context

$$LCD \vee Touch \tag{3.2}$$

where the LCD and/or the Touch feature must be selected in the variant configuration to include the mapped code. Contexts can be created by the developer at any time during the editing session and can be any presence condition over a set of features, which represent commonalities over all available variants.

A presence condition can be either a single feature condition or a feature connected with a presence condition through a logical operator. It has no length constraints and does not check for redundancy. Presence conditions can also be evaluated in terms of validity for each variant. The validity is therefore true when a presence condition is fulfilled in the configuration and can be checked by a SAT solver (s. Section 2.1.3) with the statement

$$SAT(FC \wedge \bigwedge_{f \in C_V} f \wedge \bigwedge_{f' \in F \setminus C_V} \neg f'). \tag{3.3}$$

The feature context (FC) must correspond to the configuration in terms of selected and unselected features as otherwise the mapped code can not be included into the variant. Therefore, the feature context which are usable in a specific variant are the feature contexts which are satisfiable in the variant specific configuration.

With a valid feature context, the edit is then included into the corresponding variant. This definition is similar to the definition given by Berger et al. [10], which are also defining feature-to-code mappings as presence conditions. The difference is that they need to know all presence conditions in advance while feature contexts can be declared at any time in the development process. Also, feature contexts do not need to be build up on other information sources, which means that as long as the developers understand the context they are working with, the developers can use any identification strings for features. This also implies that neither a feature model, a document structure, or an existing mapping are needed. Nevertheless, further information could improve the understandability of feature contexts. Dependencies on the basis of a feature model can deliver developers important informations for a feature context such as whether feature interactions are possible or not. It can also be used for feature context validation, where feature contexts are checked for validity in a specific variant.

Through the fact that a feature context can be added, changed, or removed at any time in the development process, feature contexts must ensure that each line can be mapped with an accurate feature context. Regardless of this advantage, removing or modifying a feature context is not an easy task as it could lead to a state where code assigned with feature mappings exists in variants but the new feature context is not valid in this variant.

3.1.2 Adding Lines

While feature contexts are used across our concept, edited lines can have three different kinds when analyzed by Myers diff algorithm [47, 48]. In this section, we want to look at what happens when lines are added to a document. For these added lines, new feature mappings are needed to be

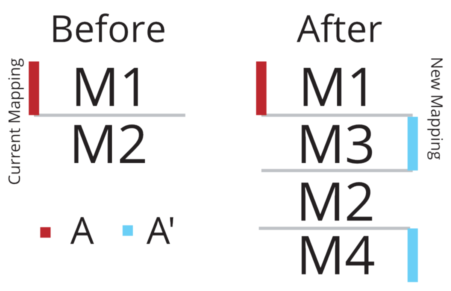


Figure 3.1: Example for adding lines. The lines $M3$ and $M4$ are added with feature context B. The resulting A' differs depending on the upper line feature mapping.

	Function		$TAUT(A \wedge B \rightarrow F_i)$	$F_i \rightarrow B$
F_0	$false$	FALSE	×	
F_1	$A \wedge B$	AND	✓	✓
F_2	$A \wedge \neg B$	A AND NOT B	×	
F_3	A	A	✓	×
F_4	$\neg A \wedge B$	NOT A AND B	×	
F_5	B	B	✓	✓
F_6	$A \oplus B$	XOR	×	
F_7	$A \vee B$	OR	✓	×
F_8	$A \downarrow B$	NOR	×	
F_9	$A \equiv B$	XNOR	✓	×
F_{10}	$\neg B$	NOT B	×	
F_{11}	$A \vee \neg B$	A OR NOT B	✓	×
F_{12}	$\neg A$	NOT A	×	
F_{13}	$\neg A \vee B$	NOT A OR B	✓	×
F_{14}	$A \uparrow B$	NAND	×	
F_{15}	$true$	TRUE	✓	×

Table 3.1: Possible boolean functions for adding lines and their respective names. A refers to the current feature mapping applied and B to the selected feature context.

	A'	
Configuration	$A \wedge B$	B
A, B	$M3, M4$	$M3, M4$
$\neg A, B$	$M4$	$M3, M4$
$A, \neg B$	$\emptyset$	$\emptyset$
$\neg A, \neg B$	$\emptyset$	$\emptyset$

Table 3.2: Influence of feature context modes on synchronizable variants when adding lines.

calculated which respect the selected feature context and fulfill the developer intentions. As those added lines do not have any further information about the current feature mapping we also include the surroundings of each added line as additional information for the feature mapping calculation. For simplification reasons, we assume that only the upper line of the surroundings are relevant to the feature mapping calculation as two different feature mappings for the upper and the lower line would lead to user interactions because of missing informations.

An example for adding lines in a document is shown in Fig. 3.1 where $M3$ and $M4$ are added under the feature context B. As $M3$ and $M4$ need feature mappings, we use boolean functions to calculate the new feature mappings. Generally, the new feature mappings can be calculated through 16 different boolean functions shown in Table 3.1 [12]. Nevertheless, many of the functions are not suitable for the calculation as they must fulfill the tautology

$$TAUT(A \land B \rightarrow F_i). \tag{3.4}$$

This is caused as the current feature mapping A and the feature context B must be valid in the variant configuration as otherwise the feature context could not be fulfilled in the variant which implements the change. Therefore, the change should not be made in the variant as it would produce dead code. This leads to a drop out for the functions F_0, F_2,F_4, F_6, F_8, F_{10}, F_{12}, F_{14} of Table 3.1 as potential calculations because they do not fulfill the tautology.

Another function constraint is the developer intention which is provided by the feature context and can be expressed through

$$F_i \rightarrow B. \tag{3.5}$$

Functions must therefore lead to a positive assignment of feature context B because otherwise the developer would have selected a different feature context. Accordingly, the remaining boolean functions are only F_1 and F_5 as they are the only ones which always lead to B which are the functions AND and B.

Projected on the calculation of the feature mapping for the lines $M3$ and $M4$, $M3$ can have either $A \land B$ or B as resulting feature mapping. $M4$ instead can only be assigned to the feature mapping B as the upper line of $M4$ does not have any current feature mapping. With this, we assign A with *true* for the calculation of $A \land B$ which leads to $true \land B \equiv B$. Therefore, the AND and B functions are the same with no upper line mapping.

After the edit is made and the feature mapping is applied, the edit can then be patched into other variants through synchronization. Table 3.2 shows the possible configuration options for all variants and whether they are possible synchronizable variants for the lines $M3$ and $M4$. The above stated equality of the functions AND and B for $M4$ is also shown in this table as $M4$ can also be

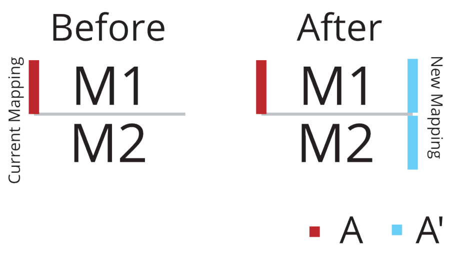


Figure 3.2: Example for removing an artifact. Both lines are removed with feature context B. The resulting A' differs depending on the selected feature context mode.

synchronized into variants that do not implement A even when the AND function was used for the calculation.

3.1.3 Removing Artifacts

Besides the addition of lines to a document, removing lines is a second case for Myers diff algorithm [47,48]. When removing lines of a document, normally a new feature mapping is not possible as the lines do not exist after removing them. Nevertheless, a mapping is still possible in our concept as the feature mappings could be synchronized into other variants. Therefore, we negate the feature context to $\neg B$ as the user decides to remove the lines explicit under the feature context B. Furthermore, the user intends also that each same line in all other variants should also be removed, if B is not implemented in the variant. Besides this, the annotation of the lines would lead to in improvement of feature mapping completeness in the other variants and therefore to a better feature traceability. Fig. 3.2 shows an example of two lines which are both removed during the edit. While $M1$ has a feature mapping assigned, $M2$ has none. Both lines are removed and do not have a feature mapping after this, but for all other variants this removal could increase the feature mapping coverage as $M1$ and $M2$ could be used to annotate the same line in all other variants which also have $\neg B$ selected. Also, $M1$ and $M2$ can be removed in all other variants where their feature mapping is fulfilled in the configuration of the variant.

For the feature mapping calculation, we also want to look at the 16 possible boolean functions [12]. As the user intends to remove the line under the feature context B, we can assign the feature mapping to other variants using the negated feature context. This leads to a different function table as shown in Table 3.1 in Section 3.1.2, because of the negated feature context. As shown in Table 3.3, the tautology with the negated feature context

$$TAUT(A \wedge \neg B \rightarrow F_i). \tag{3.6}$$

throws out half of the boolean functions and only the functions F_2, F_3, F_6, F_7, F_{10}, F_{11}, F_{14}, and F_{15}

	Function		$TAUT(A \wedge \neg B \rightarrow F_i)$	$F_i \rightarrow \neg B$
F_0	$false$	FALSE	×	
F_1	$A \wedge B$	AND	×	
F_2	$A \wedge \neg B$	A AND NOT B	✓	✓
F_3	A	A	✓	×
F_4	$\neg A \wedge B$	NOT A AND B	×	
F_5	B	B	×	
F_6	$A \oplus B$	XOR	✓	×
F_7	$A \vee B$	OR	✓	×
F_8	$A \downarrow B$	NOR	×	
F_9	$A \equiv B$	XNOR	×	
F_{10}	$\neg B$	NOT B	✓	✓
F_{11}	$A \vee \neg B$	A OR NOT B	✓	×
F_{12}	$\neg A$	NOT A	×	
F_{13}	$\neg A \vee B$	NOT A OR B	×	
F_{14}	$A \uparrow B$	NAND	✓	×
F_{15}	$true$	TRUE	✓	×

Table 3.3: Possible boolean functions for removing lines and their respective names. A refers to the current feature mapping applied and B to the selected feature context.

	A'	
Configuration	$A \wedge \neg B$	$\neg B$
A, B	$\emptyset$	$\emptyset$
$\neg A, B$	$\emptyset$	$\emptyset$
$A, \neg B$	$M1, M2$	$M1, M2$
$\neg A, \neg B$	$M2$	$M1, M2$

Table 3.4: Influence of feature context modes on synchronizable variants when removing lines.

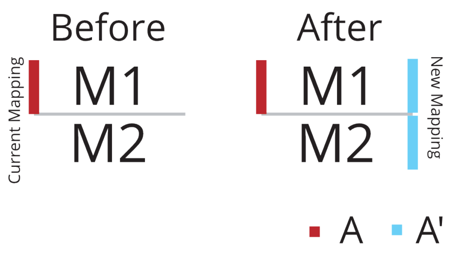


Figure 3.3: Example for changing an artifact. Both lines are changed with feature context B. The resulting A' differs depending on the selected feature context mode.

	A'	
Configuration	$A \wedge B$	B
A, B	$M1, M2$	$M1, M2$
$\neg A, B$	$M2$	$M1, M2$
$A, \neg B$	$\emptyset$	$\emptyset$
$\neg A, \neg B$	$\emptyset$	$\emptyset$

Table 3.5: Influence of feature context modes on synchronizable variants when changing lines.

remain. Also, after applying the user intention equation

$$F_i \rightarrow \neg B, \tag{3.7}$$

again only two functions remain as possible candidates for the new feature mapping. $A \wedge \neg B$ and $\neg B$ are therefore the only choices for a feature mapping where lines are removed. Additionally, this approach can also be seen as prune and tag, an inversion of tag and prune [11] as lines are pruned before the lines are tagged.

Also, synchronizing the edits is the opposite of the synchronization process for line additions as the feature context is negated. Table 3.4 shows that the lines $M1$ and $M2$ are then synchronized into a specific variant when $A \wedge \neg B$ or $\neg B$ is fulfilled in the configuration.

3.1.4 Changing Artifacts

Another operation runs when a line has changed. This could happen when developer perform a refactoring or adapt parameters to a new environment. Figure 3.3 shows an example for changed artifacts. Both lines are changed under feature context B and while $M1$ already has a mapping, $M2$ has no mapping. For the calculation of the new feature mapping, we can reuse the boolean functions for adding a line as we work with the same feature context which is not negated. The only difference lies in the calculation for $M1$ as the surrounding is replaced with the current feature mapping which leads to a change of A in the table. As the 16 boolean functions [12] are not depending on A, we can easily exchange the underlaying feature mapping as the current feature mapping and the feature mapping of the upper line are provided by the same document and are conceptually close to the same meaning. Therefore, feature mappings can be calculated through the boolean functions $A \wedge B$ and B where A is the current feature mapping and B is the selected feature context. Also, the possible synchronizable variants are not different from the one when adding a line (s. Table 3.5).

Nevertheless, in theory, handling an artifact change individually as a separate operation is quite controversial. This is caused by the reason that the operation could maybe also be done trough removing the original mapping with the negated feature context (s. Section 3.1.3) and adding the feature mapping again. Despite of that, we still decided to see a change operation as independent operation as Myers diff algorithm [47, 48] is also introducing a separate change case.

3.1.5 Feature Context Modes

We have seen in the last sections that each operation has multiple boolean functions that are possible for calculating new feature mappings for added, removed, or changed lines. As we cannot decide for the developer which function is appropriate in the current edit process, we want to give the decision to the developers in form of feature context modes. This is possible because the boolean functions are similar for all cases despite the negation for the removing case. Therefore, we can select $A \wedge B$ and B as representatives for all cases. Since A is declared as current feature mapping or upper line mapping, B represents the feature context which can also be negated for the case of removing lines from a document. Both functions seem to be reasonable as a conjunction can represent feature interactions and B builds the mapping only based on the developer selected feature context. In the following, we therefore want to refer to $A \wedge B$ as *merge* mode and describe the expression B as *override* mode for easier remembrance by the developers. Also, the removal case will negate the feature context B which leads to $A \wedge \neg B$ instead of $A \wedge B$ (s. Section 3.1.3). Equal to this, $\neg B$ is the negation of B and should therefore also be used when a negation is needed with active *override* mode.

Beside the two boolean functions, we also want to introduce the *enhance* mode to help developers to increase the feature mapping while keeping all other feature mappings. Therefore, the mode extends the *override* mode and only enables a mapping when the changed line was not mapped to a feature before. It can be used in parallel to the *merge* and the *override* mode as the current feature mapping is always true which leads to the same calculated feature mapping for both modes. It is similar to the boolean function B and should help developers in minimizing the number of switches they must undertake when working with different feature contexts. A change can then be done under the same context as the current feature mapping and is the only way to disable the inclusion of the feature context while keep recording the edits. This also implies that it is only useful to developers when lines are changed in a file (s. Section 3.1.4).

3.2 Limitations & Extended Concept

Our basic concept only takes the current feature context and the current feature-to-artifact mappings into account. Still, analyzing and calculating the best feature-to-artifact mapping possible often needs further information. As information come with effort on the developer site and our concept tries to automate as much as possible, our line-based concept does only require a list of feature contexts and configurations which could be enriched with more information in the future. With the actual used information, a general recording and automated synchronization is possible, but does not deliver the best possible product as it could be improved through structure information such as ASTs [49], which we will discuss in Section 3.2.1. Another information source can be a feature model (s. Section 3.2.2) which could be created for all variants and help building better presence conditions using the structure and dependency information of the feature set.

3.2.1 Recording with AST

As shown in the previous sections, line-based feature mappings are sometimes very complicated as a line can held multiple artifacts at once. An example is a line with a conditional statement which is composed out of different clauses. Listing 3.1 shows such a conditional statement where all clauses are implemented by different features and therefore need their own feature mapping which is not applicable with a line-based approach. Also, functions cannot be mapped to lines when features are extending the function header with parameters as shown in Listing 3.2. Beneath the function header, the function call is furthermore a problem when adding parameters to functions. Both problems could be solved with expanding the line so that each feature can be mapped to a line only containing the necessary part for the feature mapping. The problem is that this solution also lowers the readability and understandability of source code which we want to avoid.

```
if (_lcd.available() && _touch.ready()) {
    _lcd.print('Menu');
    _touch.enable();
    ...
}
```

Listing 3.1: Example feature mapping enhanced with an AST. Features *LCD* and *TOUCH* can be composed into one conditional statement.

```
void update_usb_status(const bool forceUpdate) {
    static bool last_usb_connected_status = false;
    if (last_usb_connected_status != Serial || forceUpdate) {
        last_usb_connected_status = Serial;
        write_to_lcd_P(PSTR("{R:UC}\r\n") : PSTR("{R:UD}\r\n"));
    }
    ...
}
```

Listing 3.2: Example feature mapping enhanced with an AST. Features *USB_FUPDATE* can extend function parameters and also used in conditional statements.

Another solution for this kind of problem is an improvement of granularity of the approach. While line-based mappings can be great as they are supported for all text files, source code and domain specific languages have a specified structure defined which can be used for increasing the granularity for feature mappings. This improves readability and understandability but can also be used for enhanced granularity when building abstract syntax trees (ASTs). Abstract syntax trees (s. Section 2.1.5) transform an artifact into a structured tree which could be traversed and analyzed. A tree has always a root node which represents the complete artifact and has sub-nodes for each structural child it contains.

With a complete AST, we can detect clones [4,32] or understand how the software evolves during development [49] which can also be a useful addition when developing variants. Despite that, we can also use the structure to calculate only the necessary parts of a line that has changed or retrieve the hierarchy and nested mappings of the edited part. It can be used for a deeper analysis of the dependencies of an edit. Both advantages can be used to annotate the correct parts of the line shown in Listing 3.1 as we can include the structure and dependencies in our feature mapping calculation. Furthermore, we can integrate the feature mapping of a conditional statement into the mapping of a line within the body of the conditional statement and can also be used for an automated merge when an edit is surrounded by the same feature mapping. This can ensure that artifacts are not taken over in other variants by accident because they are missing the feature mapping of the dependency.

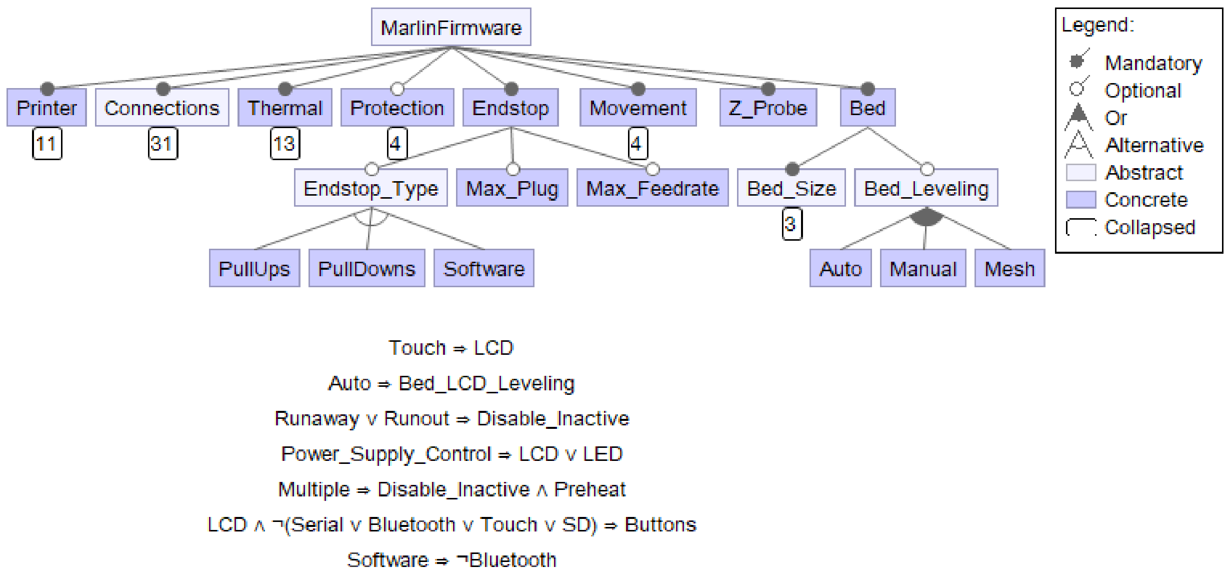
In total, ASTs could improve the feature mappings as they deliver more informations about a feature mapping. Nevertheless, an AST must be build at each edit which raises the level of complexity for determining a feature mapping and requires no syntax errors to be build. Also, ASTs are only available to structured artifacts such as source code or other domain specific languages as it would bring no advantage when a document has no structure. This is also a reason why using abstract syntax trees always need fallback solutions like line-based mapping and the potential for improvement should also be evaluated in Chapter 5.

3.2.2 Validation and Metrics with Feature Model

Beside the problem of granularity, another problem lays in choosing the best possible feature contexts because feature mappings need to be as good as possible when developers specify them. Our concept supports this by allowing creation of feature contexts whenever the developer needs to change them. These presence conditions can be built freely without any constraint in terms of features, dependencies, or constraints. Nevertheless, this leads to a problem of validation and verification of freely created feature contexts. This could lead to dead code where feature mappings could never be evaluated to *true* in the variant. Furthermore, the developer can easily miswrite a term of a feature context, so that a duplicate feature context can be created. This would also split up the feature implementation into separated groups and would decrease the feature traceability. Another point is that presence conditions can grow in complexity and length. Both of them decrease the readability and comprehensibility of the context and should be therefore avoided. The problem occurs as we do not know the relations between the different terms of the presence condition and cannot calculate the shortest possible presence condition.

As a solution for these problems, we could use a feature model like Fig. 3.4. As explained in Section 2.1.3, the feature model describes the variability points over all variants and contains features, their relations, and cross-tree constraints. Also, a feature model can be annotated with colors [28] or other attributes [5] which both help to illustrate the variability even more. With these information, feature contexts created by developers could be validated through a SAT solver. Furthermore, feature contexts and later feature mappings could be validated in terms of satisfiability [46] to prove that they do not produce dead artifacts which can be defined similar to dead code [64]. Through checking for generalisability and redundancy, we can determine a minimized version of feature contexts and feature mappings. Also, misspelled features can be prevented through the feature set of the feature model as each feature must be unique. With all this improvements, our concept could help developers in creating a good feature context and maintaining a healthy state of the artifacts and the feature mappings.

Beside solutions for both, the feature context creation and the feature mapping, we could also use the feature model for development metrics. We could check for complex feature interactions through calculating the number of features used in the feature mapping, so that developers can take a look at these mappings again. Also, we can give developers a hint when a relation between two features is possible through checking presence conditions. Developers could then merge the duplicated features, change a dependency, or create a constraint in the feature model to formalize this in the feature model. Furthermore, we could check for not used features and relate the number of mapped and unmapped artifacts, which can help to evaluate the state of conversion from clone-and-own to our concept.



Touch ⇒ LCD
Auto ⇒ Bed_LCD_Leveling
Runaway ∨ Runout ⇒ Disable_Inactive
Power_Supply_Control ⇒ LCD ∨ LED
Multiple ⇒ Disable_Inactive ∧ Preheat
LCD ∧ ¬(Serial ∨ Bluetooth ∨ Touch ∨ SD) ⇒ Buttons
Software ⇒ ¬Bluetooth

Figure 3.4: Feature model for simplification and validation feature context for the 3D printer firmware Marlin [44].

In total, a developer can improve the development process in terms of validation and metrics when using a feature model. This is possible through the fact that feature models provide information which can be used directly with the artifact mappings. With this one document, many improvements can be made and a further conversion into a software product line is also easier as a feature model is fundamental and needs to be created for the product line process. Therefore, creation and maintaining a feature model might be a good idea by the time when developers are switching to our concept or even sooner as it can enhance all possible steps when evolving the process from clone-and-own. Despite that, it is nevertheless a time consuming task which should be done by domain experts as it will decide the quality of all feature mappings hereinafter.

3.3 Summary

In this chapter, we introduced our line-based concept for feature-to-artifact mapping. This has the advantage that widely available types of artifacts are supported. We identified three operations based on myers diff algorithm [47, 48]: *Add*, *Remove*, and *Change*. For these operations we defined our feature mapping with respect to the current feature mapping, the selected feature context, and the activated feature context mode. We defined feature contexts as presence conditions and feature context modes as boolean functions which are conducted out of all possible functions for two terms. The conclusion were the *merge* and *override* mode with their negations. We also presented the *enhance* mode as a way to change code without any feature mapping modification as it ignores the feature context completely. Afterwards, we showed limitations and how an extension with structure or feature models could benefit the concept in terms of automated feature mappings. With structure available, it would be possible to further automate feature mapping for artifacts as the surrounding structure would be visible to the calculation algorithm. It could also enhance the granularity with multiple feature mappings in a single line. Otherwise a feature model could enable an improvement in terms of feature context creation as contexts could be automatically validated against the feature model. Feature contexts could also be simplified as feature models could create relations between the in our concept loose features.

4 Implementation in VariantSync

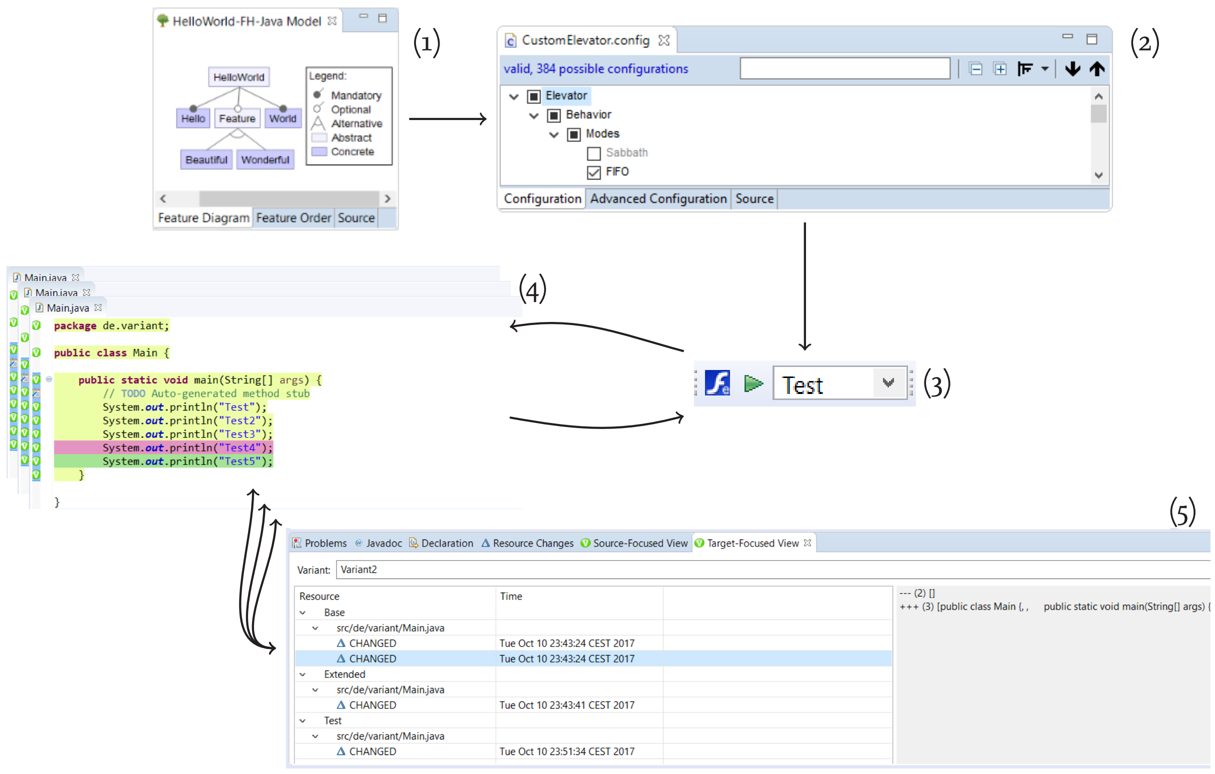
In this chapter, we want to take a look at the prototypical implementation of our concept for the automated recording of feature mappings, which we introduced in Chapter 3. Due to time constraints, we only concentrate on line-based mappings (s. Section 3.1) and not on the AST (s. Section 3.2.1) and feature model improvements (s. Section 3.2.2). We implement the concept in VariantSync [43], a tool to improve the development process of software variants.

4.1 Current Workflow and Limitations in VariantSync

VariantSync is a plug-in for the Eclipse framework [23] and provides already basic line-based feature mapping support. It is based on FeatureIDE [18,31,38,45,65], a software product-line tool which supports feature model and configuration editing. It is one of the most known tools in research and often used in prototypical development because of extensibility [33] and active maintenance [18]. Before this thesis, VariantSync prototype can override a line with the current selected feature context, calculating more advanced feature mappings are not supported. Furthermore, feature context modes (s. Section 3.1.5) are currently not yet available to the developers and merging or enhancing the current mapping is not possible.

The current process can be divided into four different parts as shown in Fig. 4.1. The first two parts are preparations for the development and are not necessary after the first initial run. First, we create a FeatureIDE project with the VariantSync composer. A composer describes the variability approach used for development and could enable individual calculations or executions of specific task when building. In our case, it also creates a basic feature model to start with. It is a support project, which contains all necessary information for all variants for the later development process. Internally, the configuration project manager (s. Fig. 4.2) will create an instance of a configuration project which holds all information for the feature mapping and the synchronization later. In detail, it will instantiate managers for feature contexts, feature mappings and patches. The feature context manager handles all related feature context actions, for example the creation of a new feature context. It also contains a list of already created contexts. Furthermore, the mapping manager gives access to all current feature mappings for all variants which are registered in the configuration project and the patches manager registers the edits with the help of diff algorithms, which is in the case of VariantSync Myer's diff algorithm [47,48]. All three managers are accessible through the configuration project and provide interfaces to implement own algorithms through Eclipse exten-

https://doi.org/10.24355/dbbs.084-201812191127-0



```java
package de.variant;

public class Main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Test");
        System.out.println("Test2");
        System.out.println("Test3");
        System.out.println("Test4");
        System.out.println("Test5");
    }

}
```

Variant: Variant2

Resource	Time
Base	
src/de/variant/Main.java	
CHANGED	Tue Oct 10 23:43:24 CEST 2017
CHANGED	Tue Oct 10 23:43:24 CEST 2017
Extended	
src/de/variant/Main.java	
CHANGED	Tue Oct 10 23:43:41 CEST 2017
Test	
src/de/variant/Main.java	
CHANGED	Tue Oct 10 23:51:34 CEST 2017

```
--- (2) []
+++ (3) [public class Main {, ,     public static void main(String[] args) {
```

Figure 4.1: VariantSync [51] workflow for developing variability before this thesis.

sion points. Alongside basic internal implementation for variability and file handling, FeatureIDE provides a feature model editor, which helps developers to adapt the basic feature model easily to the domain of the variants. Through this, a common set of features and their relationships can be build up for the second step and also a basic list of feature contexts can be generated.

In the second step, a configuration file is created in the configuration project for each variant with the help of the FeatureIDE configuration editor. The editor is feature rich and supports on-time checking for satisfiability as explained in Section 2.1.4. The configurations will be needed later for the synchronization as each configuration allows calculations if an edit can be included in the own variant or not. We also speak of synchronizable variants which means variants that also fulfill the feature mapping in their specific configuration. Furthermore, every configuration will be checked whether a corresponding project exists. If the project exists, it will be added to the list of variants in the configuration project (s. Fig. 4.2) and is considered as synchronization target for all edits. If no variant exists yet, configurations are marked with problem markers in the Eclipse framework and can be solved through an automated project creation process as it can automatically create a Java project with the configuration name. Otherwise, the nature can be added to already existing project which are imported into the workspace.

With these first two steps in the process of the workflow, the tool environment is set-up and

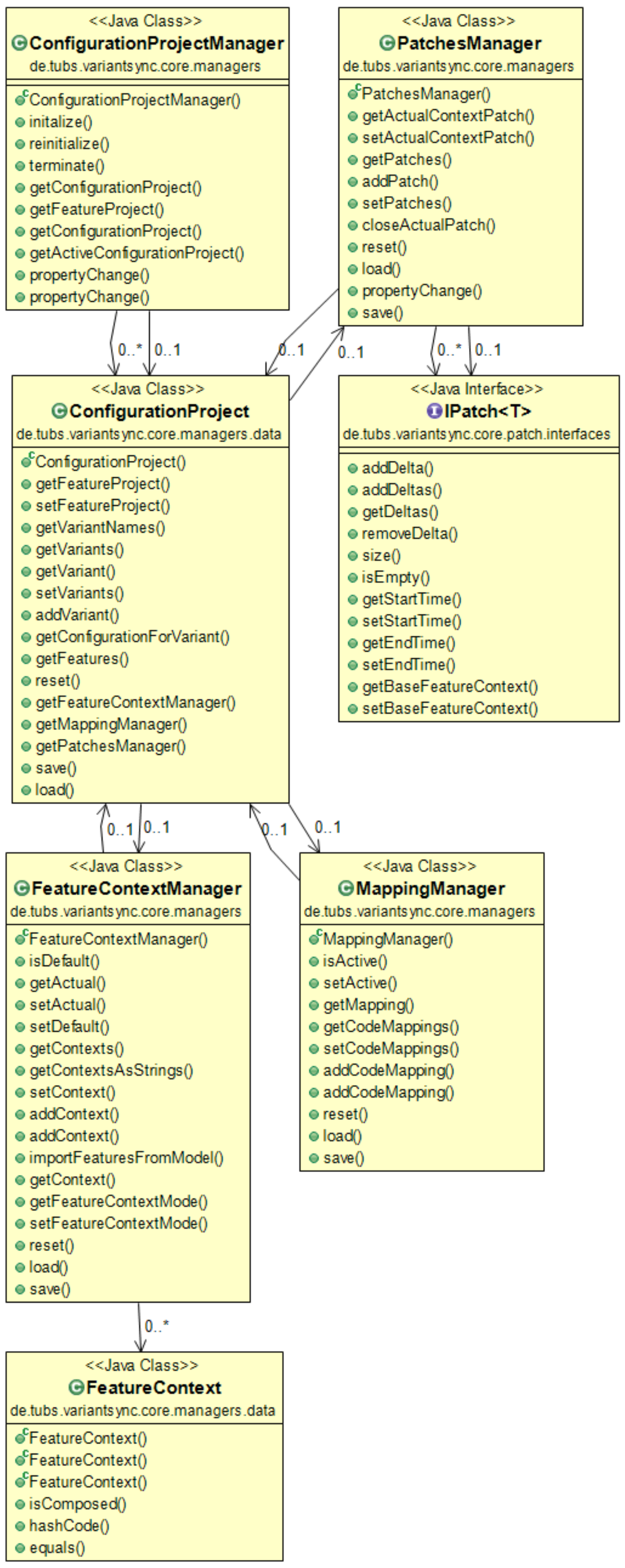
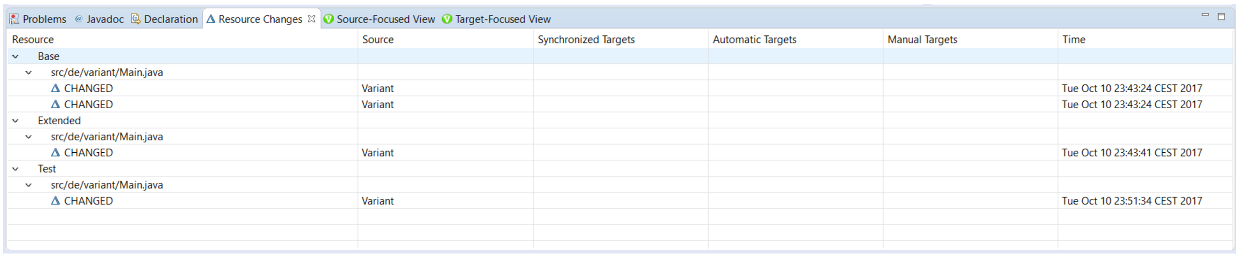
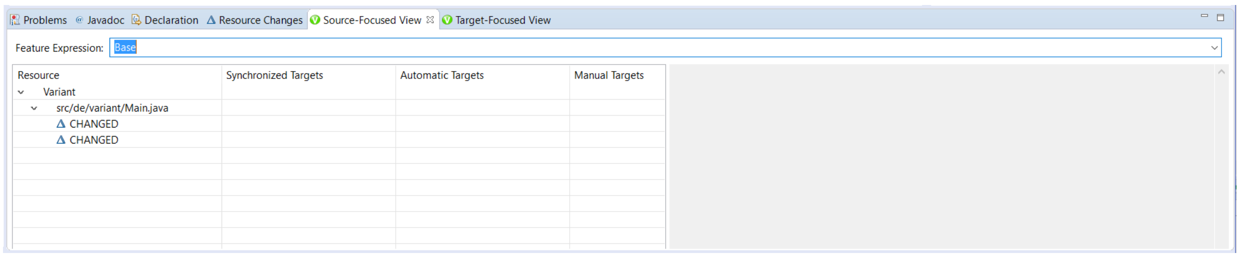


Figure 4.2: Class diagram of the manager classes implemented in VariantSync [51].



Resource	Source	Synchronized Targets	Automatic Targets	Manual Targets	Time
Base					
src/de/variant/Main.java					
CHANGED	Variant				Tue Oct 10 23:43:24 CEST 2017
CHANGED	Variant				Tue Oct 10 23:43:24 CEST 2017
Extended					
src/de/variant/Main.java					
CHANGED	Variant				Tue Oct 10 23:43:41 CEST 2017
Test					
src/de/variant/Main.java					
CHANGED	Variant				Tue Oct 10 23:51:34 CEST 2017

Figure 4.3: Resource Changes View in VariantSync [51]. Edits under different features and with only one variant present. [62]



Feature Expression: Base

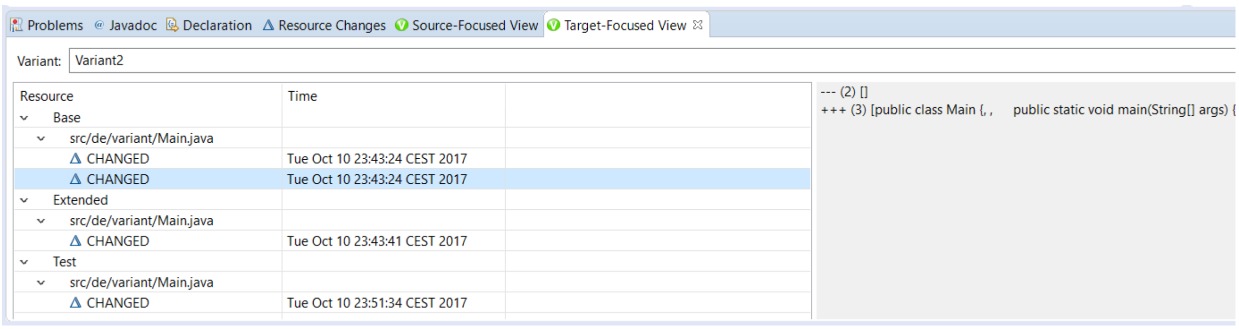
Resource	Synchronized Targets	Automatic Targets	Manual Targets
Variant			
src/de/variant/Main.java			
CHANGED			
CHANGED			

Figure 4.4: Source-Focused View in VariantSync [51]. Edits under the feature context *Base* and with only one variant present. [62]

developers can start creating source code in the variant projects which are annotated with the feature mapping. Nevertheless, it can be required that both initialization steps can be repeated at any time as otherwise a change of the feature model or any configuration would not be possible. The developer can also manually create a new feature context based on the extracted feature set from the feature model which helps when implementing feature interactions as the current process only supports the override feature context mode.

The recording is done through selecting a feature context, starting the recording and making changes to the created or imported variants. The feature context manager (s. Fig. 4.2) will hereby control the selected feature context and the mapping manager will handle the state of the mapping, which can be active or inactive. Also a patch with the current feature context is created through the patch manager, which will be filled with the edits that happen after the recording is started. Otherwise nothing will happen when developers change the source code which enables them to stop the mapping for an edit that is only variant-specific. When an edit has happened and the recording is active, the mapping manager determines the feature mapping for an edit when the developer saves the file and registers the feature mapping for the variant. After an edit is done and the recording is stopped or the feature context changed, the patch manager will also close the patch and will register the diffs as a patch for all variants to apply where the configuration is satisfiable and applicable.

As the patch is registered in the patches manager (s. Fig. 4.2), it can be synchronized in all synchronizable variants automatically or manually. An overview over all made patches is shown in the



Variant: Variant2

Resource	Time
Base	
src/de/variant/Main.java	
CHANGED	Tue Oct 10 23:43:24 CEST 2017
CHANGED	Tue Oct 10 23:43:24 CEST 2017
Extended	
src/de/variant/Main.java	
CHANGED	Tue Oct 10 23:43:41 CEST 2017
Test	
src/de/variant/Main.java	
CHANGED	Tue Oct 10 23:51:34 CEST 2017

```
--- (2) []
+++ (3) [public class Main {, ,     public static void main(String[] args) {
```

Figure 4.5: Target-Focused View in VariantSync [51]. All edits synchronizable with variant *Variant*2. [62]

resource changes view (s. Fig. 4.3). It shows the patches with their source variants and also displays the automatically and manually synchronizable variants as well as the variants which are already synchronized. With this view, the developer can see all changes and can estimate the current state of equality between all variants. If the patch cannot applied automatically, the developer must manually patch the specific variant with the edits made in this patch. Despite that, VariantSync already provides two different patch ways. First, the source-focused patch way with the source-focused view (s. Fig. 4.4) enables the developer to select patches based on their source variant. For a specific variant, the source-focused patch way shows each patch made in this variant and can be patched to all variants which are synchronizable for a patch. This can help e.g. with bug-fixing as the bug-fix only needs to be available in one variant and can be synchronized into all others. The inverse of the source-focused patch is target-focused and accessible through the target-focused view (s Fig. 4.5). It inverses the source-focused approach as it shows patches when they are available for patching in the selected variant independent of the source variant. This is useful when developers only want to update the selected variant to the latest source code and not all other variants too. After the patch is applied successfully in either way, manually or automatically, the process can start again at any step with the next recording. The feature model or a configuration can be adapted and new features implemented.

4.2 Implementation of Automated Recording and Synchronisation

While before this thesis the VariantSync workflow provides basic line-based feature mapping support, we want to extend the current implementation in the necessary parts to support automated feature mapping. A first step towards automation is to make the feature mapping itself more automated. As shown in Section 3.1, we nevertheless need support from the developers to get the best possible feature mapping. For this, we introduced feature contexts modes which work together with

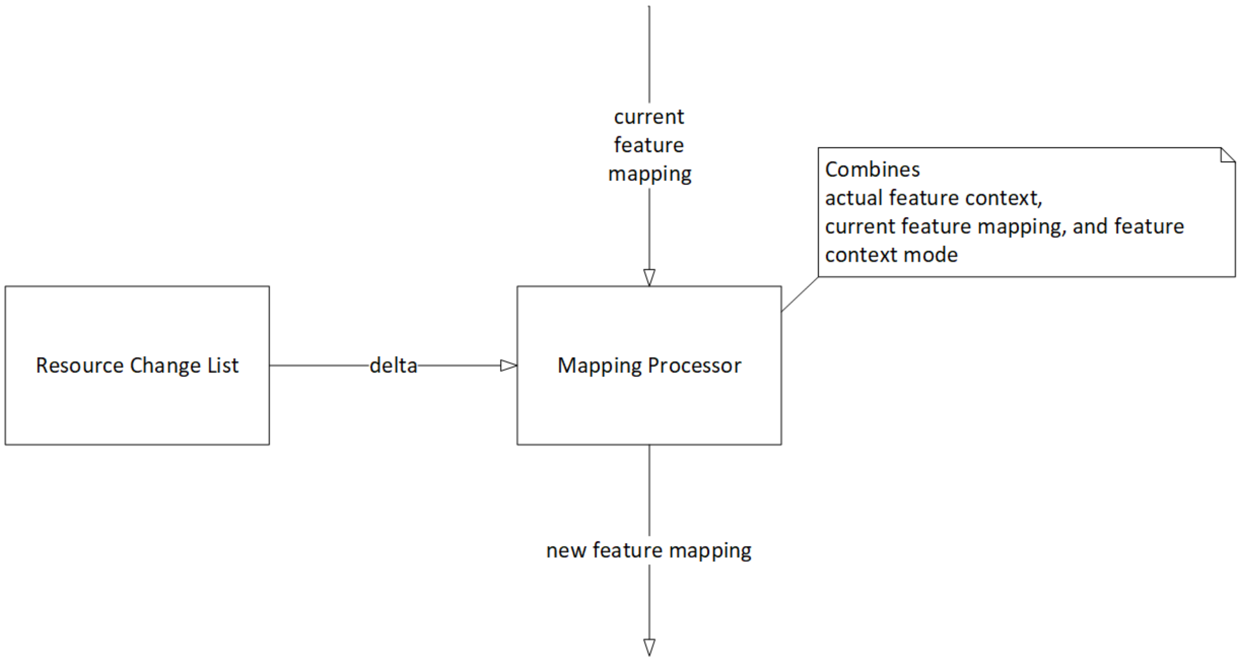


Figure 4.6: Feature mapping processor in VariantSync [51]. The new feature mappings are calculated based on the selected feature context, feature context mode and the current feature mapping.

feature contexts. Both, feature contexts and feature context modes, can be selected by the user in the toolbar of Eclipse where they are grouped beneath each other and deliver a solution for creating feature mappings based on an already existent mapping. Before, developers had to create the feature context for a line manually if the feature context was not available until then. This includes lines which contain feature interactions. With our implementation, a feature mapping can now be created automatically when a new mapping is calculated based on the selected feature context, the feature context mode, and the current feature mapping. Despite that it is also possible to have no context mode selected. With this, the developer implies that no change should be mapped, so that it could replace the start and stop functionality of the VariantSync tool before this thesis was written. It also could minimize the number of actions a developer must take to get the same result.

To realize the different feature context modes and the enhanced feature mapping at the synchronization process, the feature mapping must be changed. As a manager class should never implement functionality of its managed instances, we extended the feature mapping manager. Therewith, and also because of separation of concerns, we implemented a new feature mapping calculation processor (s. Fig. 4.6), which calculates the feature mappings for the mapping manager.

The mapping processor receives the patch made by the developer and also the former feature mapping of the edit and calculates the new feature mapping. The processor is implemented following the singleton pattern as a processor only differs for each patch approach and is instantiated

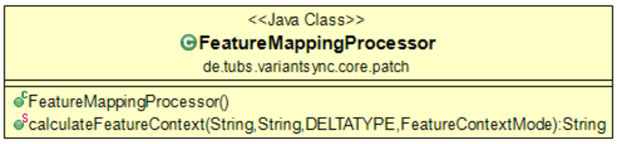

<<Java Class>>
FeatureMappingProcessor
de.tubs.variantsync.core.patch
FeatureMappingProcessor()
calculateFeatureContext(String,String,DELTATYPE,FeatureContextMode):String

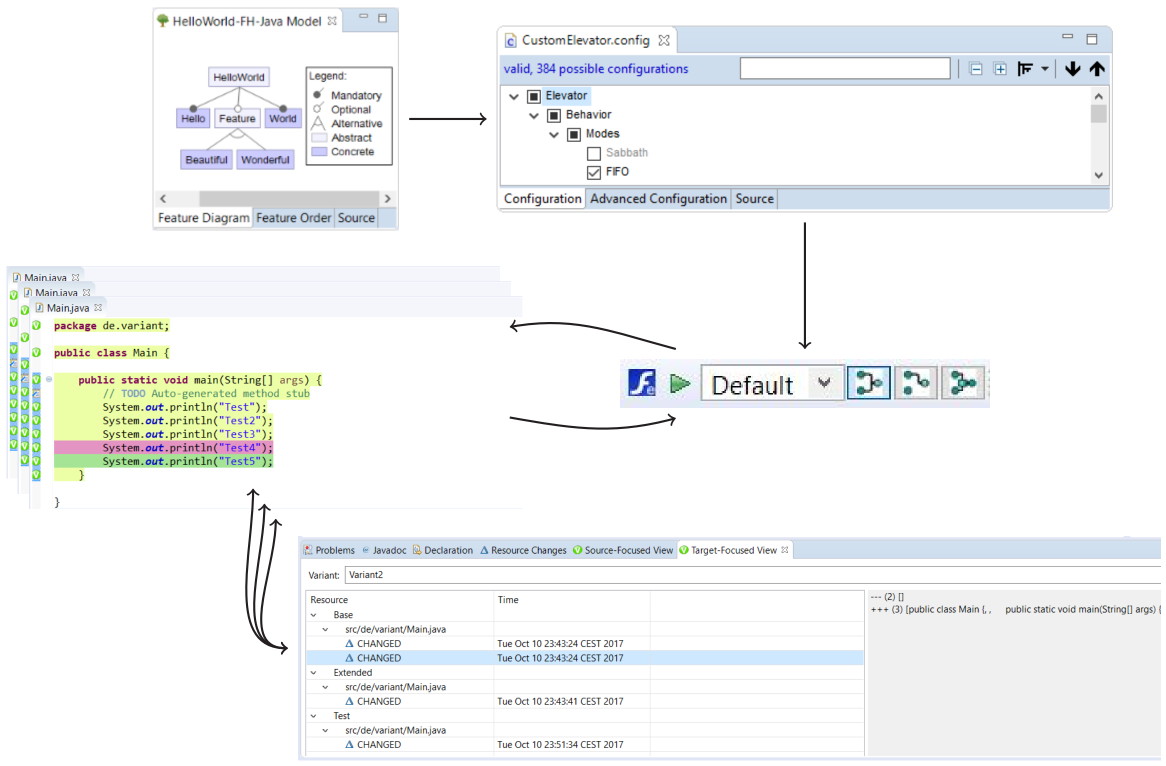
Figure 4.7: Class diagram of the feature mapping processor implementation in VariantSync [51]. Calculates the new feature mapping based on the selected feature context, feature context mode, the change type and the current feature mapping.

in the mapping manager. This comes through the fact that each approach can depend on many factors for the feature mapping calculation. One factor is the granularity of the feature mapping method of the patch approach. A feature mapping using structure can be in need for an AST for the feature mapping generation as a line can have multiple mappings. Otherwise, our processor for line-based feature mappings only needs to implement the boolean functions defined in Sections 3.1.2 to 3.1.4.

The process for our line-based concept is to check the feature context mode first, so that the right calculation method is used. Afterwards, the processor calculates the current feature mapping on basis of the boolean functions, the feature context, and the current feature mapping. The conceptual algorithm for the complete process is shown in Fig. 4.8 and for line-based mapping, the processor mainly consists of the function to calculate the feature mapping as shown in Fig. 4.7.

Despite the calculation while recording the edits, we also need to calculate new feature mappings when synchronizing variants. The mapping processor can also be used for this, as there is no difference whether the calculation happens during a recording in a variant or during a synchronization between two variants.

Another step to automation is the ability to calculate if a synchronization is possible for two variants and if the selected feature context satisfies the configuration. For the synchronization, it must be ensured that the feature mapping of the source variant is transferred into the synchronized variant correctly. For this, we also need to include a function into the patch manager to check two presence conditions in terms of satisfiability as not every feature of the feature mapping could be implemented in the synchronized variant.The satisfiability for two presence conditions can be calculated through validity checks. With this, we can be sure that each synchronization is valid and can be transferred into the other variant. This helps to keep the patches in synchronization views small and improves the overview over the different views and their possible synchronizable variants. To check the satisfiability for feature contexts, we also extended the feature context manager, which checks the context against the configuration of the variant. This also relies on a satisfiability problem and therefore, we can use the same algorithm, but with different parameters. While the patch manager accepts two configurations, the feature context manger accepts a feature context and



valid, 384 possible configurations

- [■] Elevator
  - [■] Behavior
    - [■] Modes
      - [ ] Sabbath
      - [✓] FIFO

```
package de.variant;

public class Main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Test");
        System.out.println("Test2");
        System.out.println("Test3");
        System.out.println("Test4");
        System.out.println("Test5");
    }

}
```

Variant: Variant2

Resource	Time
Base	
src/de/variant/Main.java	
CHANGED	Tue Oct 10 23:43:24 CEST 2017
CHANGED	Tue Oct 10 23:43:24 CEST 2017
Extended	
src/de/variant/Main.java	
CHANGED	Tue Oct 10 23:43:41 CEST 2017
Test	
src/de/variant/Main.java	
CHANGED	Tue Oct 10 23:51:34 CEST 2017

```
--- (2) []
+++ (3) [public class Main {, ,     public static void main(String[] args) {
```

Figure 4.8: VariantSync [51] workflow for developing variability implemented in this thesis.

a configuration.

As we need a satisfiability checker for the calculations of the synchronizations and the feature contexts, we use the boolean satisfiability solver Sat4j [57]. It is used for calculations that involve feature contexts and other boolean expressions and is already included in Eclipse [23] and also used by FeatureIDE [18] to solve boolean satisfaction problems in feature models and configurations. We can therefore conclude the validity of a synchronization as the SAT instance must have a valid assignment for the equation Eq. (3.1) from Section 3.1.

4.3 Summary

In this chapter, we first explained the VariantSync project and what was already implemented for supporting feature mappings. Before, the feature mapping process in VariantSync was limited to a line-based overwriting and had no support for automated recording of feature mappings. We implemented a new mapping processor, which handles all mapping calculations for the mapping manager and integrated the functions for selectable feature context modes. With this, we should be able to record feature mappings automatically with minimized developer interactions. In the next chapter, we will evaluate the new implementation based on our concept and investigate the introduced research questions.

5 Evaluation

In this chapter, we want to evaluate our concept for automated feature recording in terms of usability and practicability against the Marlin Firmware. As already mentioned in the chapters before, the following research questions are important for our concept as they measure the usability of the feature context modes and the degree of automation we can achieve. We also want to investigate if ASTs or feature models could enhance our concept and whether the feature mappings grew or not.

RQ1 How often feature context modes are applicable?

RQ2 Which degree of automation can be achieved when synchronizing edits?

RQ3 Can ASTs and FMs increase the automation degree?

RQ4 What is the structure of feature mappings?

5.1 Evaluation Method

Our research questions RQ1-RQ4 are mostly based on efficiency and applicability of our concept, therefore we want to evaluate the concept against a real world development project which faces everyday problems when developing with variability. For the best possible evaluation, we need a mixture of a clone and a product line. We need the possibility to generate different variants with history of the files as well as some sort of feature mapping. We also need some sort of knowledge whether the change is made only for the specific variant or should be made available in all variants. Also synchronizable variants must be determinable to calculate the applicability using some sort of configuration. For this, we selected the Marlin firmware [44] as best possible candidate as it implements 3D printer firmwares for different machine platforms. Whether code is included or not depends on these configurations as the project uses the C preprocessor to generate variants. Also a history is provided through the git version control system provided by Github [24] and code for all variants are integrated in the main code base through already closed pull-requests. Currently the Marlin firmware has over 10.000 commits and more than 350 contributors in 24 releases.

For the evaluation, we want to use the current development branch for version 2.0 ('bugfix-2.0.x'), as it is a refactored version with numerous new features being developed using pull-requests. As test subjects, we inspect the 11 lowest numbered pull-requests with 229 individual patch operations adding all kinds of different functionality from LCD panel support to the implementation of new

https://doi.org/10.24355/dbbs.084-201812191127-0

commands for 3d printers and therefore tagged with the “PR:New Feature” tag. We also want to simulate the impact of these pull requests and their operations on six possible variants for 3d printers. Four printers are widespread all over the world, one is the standard development configuration and one is a delta printer to represent a wide range of printers. As furthermore each printer needs some extra features enabled as most options are only optional features, we want to define the following additions to the variants:

1. **Default**, which has an ultra LCD, three fans with higher speed, and a USB connection.
2. **Creality CR-10S**, which has the CR-10 stock panel, two fans, auto bed leveling, SD card support, and an advanced pause feature. The printer also supports babystepping and EEPROM.
3. **Creality Ender 4**, which has a minipanel, only one fan, two serial connections, and supports also auto bed leveling, SD cards and EEPROM.
4. **SCARA**, which has three fans and two serial connections not including the usb connection.
5. **Anet A8**, which has a 1602 display, two fans with a higher fan speed, a second Z-axis stepper motor, babystepping, SD card support and EEPROM.
6. **Kossel Pro**, which has one fan with higher fan speed, two serial connections, and supports SD cards.

With these variants defined, we calculate the new feature mapping for each patch through the process described in Chapter 3. We also note which mode for each patch could be used to get the new feature mapping and in which variants the patch can be synchronized with and without conflicts. Despite the fact that we implemented our concept in VariantSync [51] , we had to validate our concept by hand as the variant generation in marlin was not possible. This was caused by the implementation of the Marlin firmware [44] product-line as the variability was implemented through macro preprocessor directives which could not be easily processed without also expanding all macros, which would make it impossible to reconstruct the pull-requests for our evaluation.

5.2 How Often Feature Context Modes Are Applicable?

Feature context modes should help developers in minimizing the number of modifications of the feature context. Nevertheless, feature contexts in combination with overriding the feature mapping could bring a better result. Also feature context modes could maximize the development effort when the modes must be switched continuously. Therefore, we want to evaluate with this research questions if feature context modes are sufficient for developers and enhance their development process. For the evaluated patch operations, Table 5.1 shows the number of times a mode can be used to create the correct mode. While only six of 299 times no mode could be applied, enhance mode

	Merge	Override	Enhance	None
Unchanged	82	82	82	
Overridden		19		
Merged	55	55		
Newly Annotated	67	67	67	
Not Mapped				6
Sum	204	223	149	6

Table 5.1: How often feature context modes can be used for all 229 operations broken down by the patch operations which were performed during the evaluation?

can be used in 65.07% of the cases. Furthermore, the merge case could be used in 89.08% to apply the correct feature mapping and the override mode could be applied at every patch which accepts a mode (97.38%). This is due to the fact that we could override the feature mapping with the correct feature context selected each time. As the research question is the degree of automation achieved through feature context modes, we can easily see that a majority of cases can use the merge context mode which concatenates the feature context and the current feature mapping automatically. This is achieved through the high variability in the feature mapping of the Marlin firmware [44] as the C preprocessor and many feature mappings not change even when new functionality is implemented. This is also shown by the huge amount of cases where the enhance mode can be used as the mode does not change the feature mapping if a feature mapping is already present. Nevertheless, the mode can be used to minimize the time for changes where the feature mapping should not be changed. When using the merge mode, quite complex feature context with three and more feature interactions (s. Fig. 5.3) can be time-consuming to support in the feature context selection. With these results, we can also conclude a standard case when working with feature context modes. As already mentioned before, the merge mode can be used in about 90% of the cases and is also available in 54.71% (without unchanged feature mappings) of all cases when a mode can be applied. This allows the automated feature mapping with much less hassle as with the override feature context mode, where each feature context must be explicitly selected even although the override mode could be used in every case.

5.3 Which Degree of Automation Can Be Achieved When Synchronizing Edits?

The second research question we want to evaluate is the question of how often changes can be synchronized without conflicts. Beneath this question, we also gain information about the possible degree of automation which is a central goal for our work. For this, we evaluated for each operation whether it could be integrated or not for each of the six variants. As expected, most patches are not

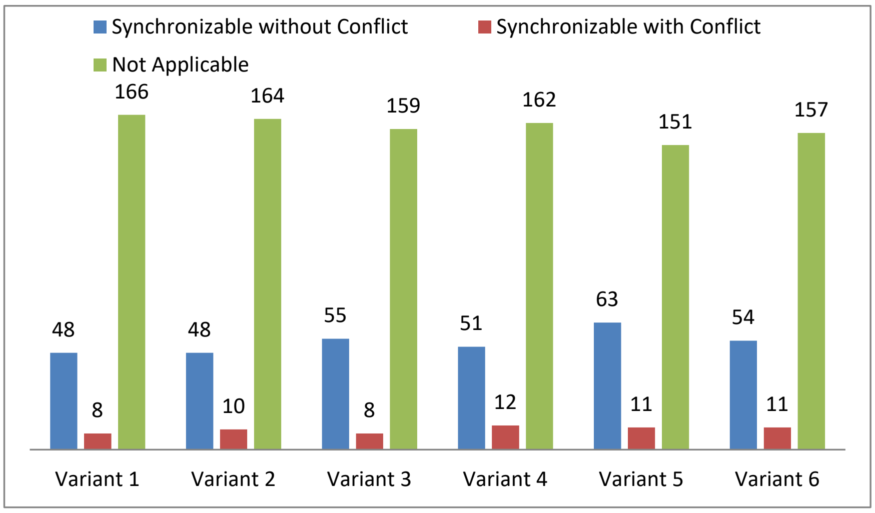
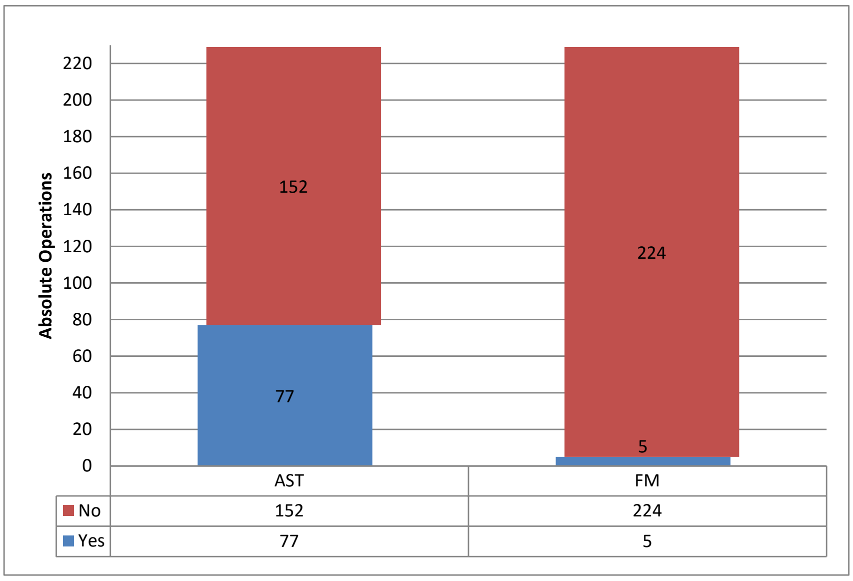


Figure 5.1: How often patches can be applied to variants and if they can be synchronized automatically without conflict when they can be applied?

applicable for specific variants. This is a result of the configurations as no configuration implements all features and we can therefore conclude that mostly very specific features are implemented or changed. Despite that, the statistics shows that out of 60 to 70 possible synchronization only 12.70% to 19.05% are in conflict. In total, this shows that patches can often be applied automatically without any user interactions.

5.4 Can ASTs and FMs Increase the Automation Degree?

As our concept is a line-based approach without any need for additional information from the file itself, it is critical that we can calculate the best possible feature mapping. Nevertheless, often additional information could be of help to automatically create a feature mapping which match the developer intents. ASTs and feature models may provide these information with the calculation as explained in Section 3.2, but the question is how often additional information are helpful. Figure 5.2 shows that the feature model could only be used in 2,18% of the cases. This can be explained through the fact, when looking at the feature model of the Marlin firmware. Most features are optional and do not have any hierarchy which results in the problem that the feature model is not helpful in almost all cases. Despite that, it can help when creating new feature contexts as it can minimize the amount of features by removing features which are redundant through duplication or hierarchy. In contrast, the AST could be helpful in 33,62% as the surrounded feature mapping



	AST	FM
🟥 No	152	224
🟦 Yes	77	5

Figure 5.2: How often ASTs and feature models can be helpful by the feature mapping calculation for each of the 229 operations?

could be determined without a user interaction so that it could be used when calculating the new feature mapping. With this, feature context modes could be automatically evaluate the document feature mappings and structure so that a third of all feature mappings could be calculated without the user selecting a feature context. Still, this share might be to small to implement ASTs for all possible types of files. Therefore, in conclusion we can say that in this particular case, the feature model does not help with any decisions and a AST only could help in a small amount of cases as it would help in automation.

5.5 What Is the Structure of Feature Mappings?

As the feature mappings should enable a better feature traceability and software reliability, developers should also understand feature mappings without problems. Nevertheless, feature mappings can get complex in terms of feature interactions after some time of development the same lines. Complex feature mappings introduce a higher risk for unwanted behavior [1] and should therefore be minimized. For this, we examine how many feature mappings consist of different features and when there were multiple, how many of them are needed to express the feature mapping. Fig. 5.3 shows how often a feature mapping consists out of up to six features. While feature mappings with one or two features are common in 61,75% of all cases are not surprising as feature interactions happen when implementing new functionality, it is surprising that 229 operations almost double the

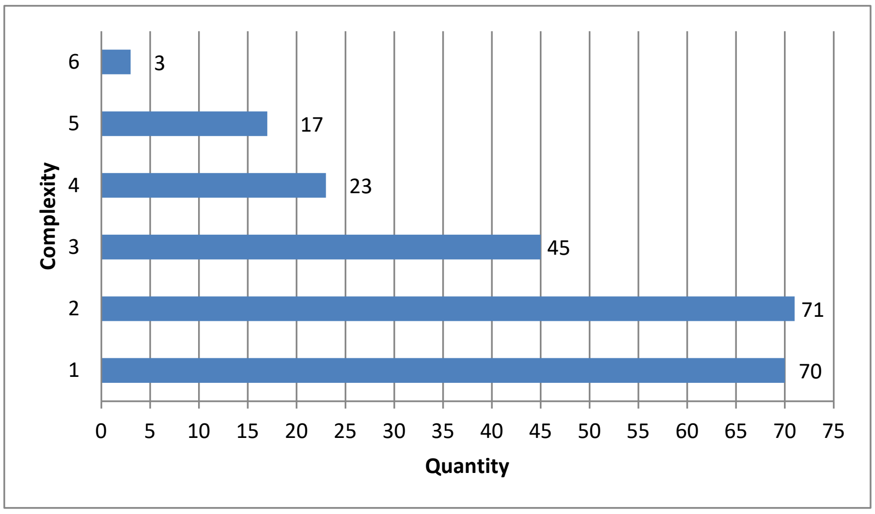


Figure 5.3: How complex are feature mappings after all 229 operations are performed?

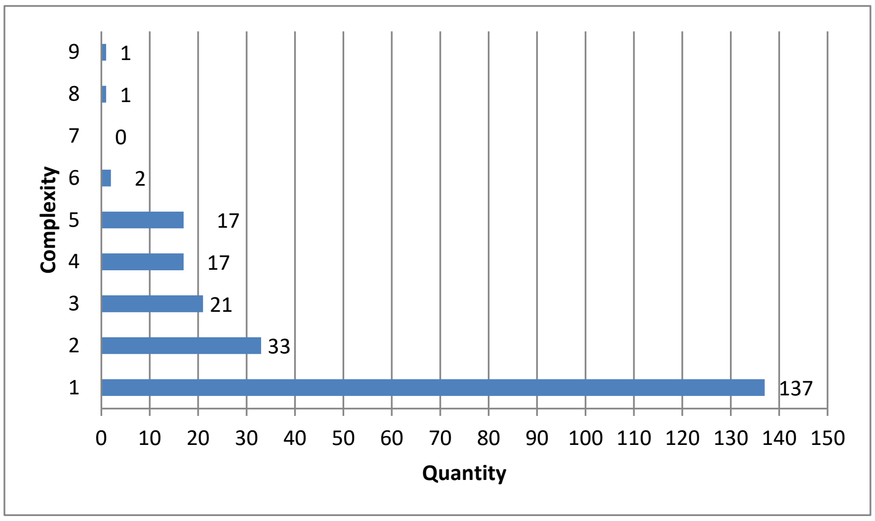


Figure 5.4: How complex were the feature mappings before all 229 operations are performed?

number of feature mappings which contain two and three features as shown in Fig. 5.4. Also, the feature mappings with a complexity of four features are increasing slightly whereas feature mappings with more than four feature interactions are on a same level or even decreasing. The outliers with eight and nine features disappear after the changes are made as these both cases could be expressed through a simple features which groups the options beneath it, instead of a preprocessor statement containing all different possible numerical options. Therefore, we can say that the resulting feature mappings get more complex over time. Despite that, in our case study feature mappings often have less than 4 features which is still a reasonable amount for developers to understand the feature mappings.

5.6 Threads to Validity

The evaluation has several points which could cause effects on the validity. In the Marlin firmware [44], often macros define values for features which are not supported by VariantSync [51]. Furthermore, the feature configurations are widely spread and not all collected in the configuration files in contrast to our first assumption. Therefore, we validated our concept by hand where each change was analyzed in terms of current feature mapping, new feature mapping, synchronizable variants, unsynchronizable variants, and usable modes. The feature context which is used to calculate the new feature mapping is selected by manually hand as in reality the developer also chooses the feature context. For the evaluation, this could lead to problems as humans always can make errors.

Another problem could be that pull requests often also define behavior for the case that features are not implemented. This could interfere with the results of the evaluation as this leads to the case that often all variants are implementing a change whether they include the feature or not. Despite that, we wanted to include the pull-requests as complete as possible and only removed changes which would also be removed trough the preprocessor. Furthermore, the variants were extracted from the same source, which could lead to the problem that source code does not have any differences to the source code in other variants. This could also imply a influence to the results of the evaluation as changes could be made to many variants as they would in real world developments.

Additionally, the feature model of the Marlin firmware has almost no hierarchy and most features are optional. A different evaluation subject instead the Marlin firmware could therefore lead to more cases in which the feature model could be useful.

5.7 Conclusion

As the above sections have shown, the concept works in our evaluation. Our modes can be used for almost all changes. While the override mode can clearly be supported anytime, the merge mode would be a good standard case as it supports many, but not all operations. Furthermore, the evaluation has shown that the enhance mode has a big impact as it shortens the time to switch between many feature contexts. Otherwise, difficult and complex feature contexts must be set before making a change. Nevertheless, our concept should be enhanced with a form of basic mode, so that also comments and general functionality can be provided to all variants. Our concept in Chapter 3 also mentioned how we could extend our concept with ASTs and feature models, but the evaluation has shown that only the ASTs could be an improvement to our line-based concept as a third of all operations could gain information from them. The feature model on the other side is not a big enhancement in this product line as the feature model of the Marlin firmware [44] is not optimal in terms of feature hierarchy and mandatory features. Another conclusion we can take out from the evaluation is that automated synchronization between variants is possible in most cases. Therefore, our concept does not only work for feature mapping, but also for the synchronization. The feature mappings often contain not more than four features, which should be a good complexity where developers are still able to understand the feature mapping.

6 Related Work

Between the extremes clone-and-own variant development and software product-lines approaches exist in research for feature mapping and feature composition. Nevertheless, our approach records feature mapping during the evolution of source code which is unique in literature. Despite this, we want to mention the following similar concepts:

FeatureMapper

Heidenreich et al. [25] introduce FeatureMapper as a tool that allows feature mapping for Ecore-based languages. The feature mapping is done through linking features to feature implementations and can be created automatically or manual. The automatic mapping can hereby map all changes to a previously selected feature which can also contain conjunctions or other logical combinations. It differs to our concept as changes to elements are changed in the mapping and are not overridden as a similar implementation of feature context modes are missing. FeatureMapper also supports visualizations in terms of filters and coloring in graphical editors which use the Graphical Editor Framework (GEF) and can additionally create model variants for different configurations.

CIDE

CIDE, developed by Kaestner et al. [35], is an Eclipse plug-in for virtual separation of concerns. In this process, source code is annotated with features in form of colors. It differs from our concept as the mapping only happens after the edit where made and not during the source code changes. Another difference is the usage of ASTs which allows fine granular feature mappings as shown in Fig. 2.4. The granular feature mappings are build on top of features from a feature list instead of feature contexts and do not support propositional logic.

ECCO

Another tool is called ECCO, which is developed by Fischer, Linsbauer et al. [21] and is an abbreviation for Extraction and Composition for Clone-and-Own. As its name stated, ECCO can extract feature code into a database with the knowledge of the configuration and all artifacts. Therefore, it only works when a variant has all necessary implementations as otherwise incomplete features are composed in the later process. At the composition process, the database is then used to generate a new variant based on a configuration. Nevertheless, some manual edits must be made when a feature interaction is new to ECCO. This in total differs from our concept as we do not focus on generation but on recording edits. Nevertheless, ECCO has many tools that helps developers develop variant-rich products. Also the database aspect is quite similar to the feature mappings manager in VariantSync [51].

Tag and Prune: A Pragmatic Approach to Software Product Line Implementation

Tag and prune, developed by Boucher et al. [11] introduces a toolchain for an annotation-based approach to develop variability. Hereby, C code blocks can be annotated with comments to signalize the feature they are belonging to, which can be removed when generating variants that do not implement the feature. The difference to our concept is based on this as well as we do not need to alter source code to annotate them with feature mappings. Furthermore, the approach does not support any automation for feature mappings as each comment must be written to the corresponding block which is calculated through an AST. Despite that, tags are more flexible as any code block can be annotated instead of line-based mappings only. Also, the approach is IDE independent as developers can use any code generator with any code editor.

Computer-aided Clone-And-Own

Lapeña et al. [36] describe an approach where requirements given in natural language are used to generate new variants. Therefore, the computer-aided clone-and-own (CACAO) algorithm calculates and ranks the relevancy for all methods in already available variants. The generation can then use these results to build new variants with cloning the most relevant methods. Besides the different purposes when comparing our concept with CACAO, the granularity is on method level is implemented using keywords where keywords can be seen as features extracted from the requirements. Furthermore, the approach is extractive and does not support a direct mapping when making edits to source code which is also quite different to our approach.

A Prototype-based Approach for Managing Clones in Clone-and-Own Product Lines

Rabiser et al. [54] introduced an approach for cloning software at different level of granularities. They also implement their tool as extension of FeatureIDE [18] and can clone prototypes of products, components, and feature through feature mapping. Furthermore, they provide a static code analysis for impact determinations of edits. They support adding, removing and modifying for compositions of products, features of components, and code mappings of features. Nevertheless, they do not support calculations of feature mappings with new feature contexts but notify developers when feature mappings create problems for the cloning process.

7 Conclusion

In industry and research, most developers use clone-and-own or software product lines to implement variability as they are widely known and actively researched. But many tools and approaches exist which can be classified between clone-and-own and software product-lines as both techniques have negative points. While clone-and-own is unable to easily synchronize changes between variants, software product lines have a worse cost-benefit ratio. Therefore, we presented an approach for synchronizing clone-and-own variants which provides automated recording functionality for feature mappings to keep track about variant evolution. Automated recording of feature mappings during evolution is a big task where practicability and usability are one of the key aspects when developing concepts. Our approach has shown that it is capable of automated recording and synchronization between different variants. Our evaluation has also shown that our line-based concept does not necessary need further information about a change, but information can enhance the process in terms of automation. Artifacts can be added, removed, or changed without many adoptions in the developers way of editing as they only need to adapt the feature context and the feature context mode. Regarding other available tools (s. Chapter 6), our concept has shown that the usage of feature context modes can bring an significant advantage instead of selecting the correct feature mapping every time which is often used by other approaches. Our proposed standard feature context mode (*Merge*) can be applied in about 90% of all edit cases and out of 229 patches 12% - 19% can be automatically synced in other variants based on the variant configuration.

In total, annotation-based feature mappings, which are recorded automatically and can also be synchronized in an automatic manner to a selection of variants, can provide a huge improvement in discussions on the topic whether to choose clone-and-own or software product lines.

8 Future Work

Despite the fact that our concept works quite well, we can see some potential in different approaches and improvements which we want to discuss in this chapter.

Extended Feature Model

A first enhancement for our implementation could be the usage of an extended feature model [6] which enhance a feature model with additional parameters like attributes. In our example, the Marlin firmware [44] has many preprocessor statements with numerical or character values which is quite difficult to transform into feature mappings when the variability is build on top of feature models.

Default Feature Context

Also a default feature context should enable the implementation of general functionality and general comments as these changes often need to be synchronized into all other variants. In theory, the top-level feature in the feature diagram could be used for these changes, but it should be a constant expression without any relation to the feature model as developers should easily spot a default context as default. A default color would also improve the feature mapping readability. An open question is whether a default feature context should be selectable over the feature context selection or over the deselection which should be investigated in future.

Removing Feature Context

Another problem with feature contexts is the unknown behavior for the case when feature contexts are removed. It is still an open research question whether feature mappings are updated accordingly or stay untouched. As many different ways exist, a survey or other evaluation methods should be applied to verify the best possible behavior because it could possibly remove artifacts in all variants.

Synchronizing Feature Mappings without Edits

Our concept proposes to synchronize feature mappings with the synchronization itself. Nevertheless, feature mappings could also be applied automatically to all variants which implement the feature mapping as well. This would improve the feature mapping coverage and traceability, but is not an easy task because parallel development often causes a different document structure for each variant.

https://doi.org/10.24355/dbbs.084-201812191127-0

Removing Artifacts

Removing artifacts takes an important role in our concept as we can conclude mappings for other variants. Nevertheless, we can see further improvements as other variants could also probably gain more feature mapping coverage. Also the old feature mapping could improve some aspects of the algorithm.

Colors

Furthermore, colors could be imported with the feature model as well, so that the initial import of the features is enough to start without further action by the developer.Also it should be investigated if colors for the feature mapping could be calculated automatically.

Bibliography

[1] I. Abal, C. Brabrand, and A. Wasowski. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 421–432, New York, NY, USA, 2014. ACM.

[2] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stănciulescu, A. Wąsowski, and I. Schaefer. Flexible Product Line Engineering with a Virtual Platform. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 532–535, New York, NY, USA, 2014. ACM.

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

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 368–377, Washington, DC, USA, 1998. IEEE.

[5] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated Reasoning on Feature Models. In *Proc. Int'l Conf. Advanced Information Systems Engineering (CAiSE)*, pages 491–503, June 2005.

[6] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010.

[7] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. A First Step Towards a Framework for the Automated Analysis of Feature Models. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 39–47, Washington, DC, USA, 2006. IEEE.

[8] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wąsowski. Three Cases of Feature-Based Variability Modeling in Industry. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*, pages 302–319. Springer, 2014.

[9] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A Survey of Variability Modeling in Industrial Practice. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 7:1–7:8, New York, NY, USA, 2013. ACM.

[10] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wasowski. Feature-to-Code Mapping in Two Large Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 498–499, Berlin, Heidelberg, 2010. Springer.

[11] Q. Boucher, A. Classen, P. Heymans, A. Bourdoux, and L. Demonceau. Tag and Prune: A Pragmatic Approach to Software Product Line Implementation. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 333–336, New York, NY, USA, 2010. ACM.

[12] F. M. Brown. *Boolean reasoning: the logic of Boolean equations.* Springer Science + Business Media, New York, NY, USA, 2012.

[13] S. Bühne, G. Halmans, and K. Pohl. Modelling dependencies between variation points in use case diagrams. In *Proc. Int'l Working Conf. Requirements Engineering: Foundation for Software Quality (REFSQ)*, volume 3, pages 59–69, Klagenfurt/Velden, Austria, June 2003.

[14] P. Clements and C. Krueger. Point / Counterpoint: Being Proactive Pays Off / Eliminating the Adoption Barrier. *IEEE Software*, 19(4):28–31, July 2002.

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

[16] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* ACM/Addison-Wesley, New York, NY, USA, 2000.

[17] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An Exploratory Study of Cloning in Industrial Software Product Lines. In *Proc. Europ. Conf. Software Maintenance and Reengineering (CSMR)*, pages 25–34, Washington, DC, USA, 2013. IEEE.

[18] FeatureIDE Contributors. FeatureIDE. Website. Available online at `https://github.com/FeatureIDE/FeatureIDE`; visited on September 1, 2018.

[19] D. Fey, R. Fajta, and A. Boros. Feature modeling: A meta-model to enhance usability and usefulness. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 198–216. Springer, 2002.

[20] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *Proc. Int'l Conf. Software Maintenance and Evolution (ICSME)*, pages 391–400, Washington, DC, USA, 2014. IEEE.

[21] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 665–668, Piscataway, NJ, USA, 2015. IEEE.

[22] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. A Vision for Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. *Lecture Notes in Informatics (LNI)*, pages 95–96, 2016.

[23] E. Foundation. Eclipse. Website. Available online at `https://www.eclipse.org/`; visited on September 1, 2018.

[24] GitHub Inc. Github - the world's leading software development platform. Website. Available online at `https://github.com/`; visited on September 1, 2018.

[25] F. Heidenreich, J. Kopcsek, and C. Wende. FeatureMapper: Mapping Features to Models. In *Companion Int'l Conf. Software Engineering (ICSEC)*, pages 943–944, New York, NY, USA, May 2008. ACM. Informal demonstration paper.

[26] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[27] K. C. Kang, J. Lee, and P. Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, 19(4):58–65, 2002.

[28] C. Kästner. CIDE: Decomposing Legacy Applications into Features. In *Proc. Int'l Software Product Line Conf. (SPLC)*, Washington, DC, USA, 2007. IEEE.

[29] C. Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, 2010.

[30] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824, New York, NY, USA, 2011. ACM.

[31] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: A Tool Framework for Feature-Oriented Software Development. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 611–614, Washington, DC, USA, May 2009. IEEE. Formal demonstration paper.

[32] R. Koschke, R. Falke, and P. Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 253–262, Washington, DC, USA, 2006. IEEE.

[33] S. Krieter, M. Pinnecke, J. Krüger, J. Sprey, C. Sontag, T. Thüm, T. Leich, and G. Saake. FeatureIDE: Empowering Third-Party Developers. In *Proc. Int'l Software Product Line Conf. (SPLC)*, New York, NY, USA, 2017. ACM.

[34] C. W. Krueger. Easing the Transition to Software Mass Customization. In *Proc. Int'l Workshop Software Product-Family Engineering (PFE)*, pages 282–293, Berlin, Heidelberg, 2002. Springer.

[35] C. Kästner. CIDE project. Website. Available online at `https://ckaestne.github.io/CIDE/`; visited on September 1, 2018.

[36] R. Lapeña, M. Ballarin, and C. Cetina. Towards Clone-and-Own Support: Locating Relevant Methods in Legacy Products. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 194–203, New York, NY, USA, 2016. ACM.

[37] R. Laqua and P. Knauber. Configuration management for software product lines. In *Proc. Workshop Deutscher Software-Produktlinien*, pages 49–53, 2000.

[38] T. Leich, S. Apel, L. Marnitz, and G. Saake. Tool Support for Feature-Oriented Software Development - FeatureIDE: An Eclipse-Based Approach. In *Proc. Workshop Eclipse Technology eXchange (ETX)*, pages 55–59, New York, NY, USA, 2005. ACM.

[39] D. Lettner and P. Grünbacher. Using Feature Feeds to Improve Developer Awareness in Software Ecosystem Evolution. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 11:11–11:18, New York, NY, USA, 2015. ACM.

[40] L. Linsbauer, F. Angerer, P. Grünbacher, D. Lettner, H. Prähofer, R. E. Lopez-Herrejon, and A. Egyed. Recovering Feature-to-Code Mappings in Mixed-Variability Software Systems. In *Proc. Int'l Conf. Software Maintenance and Evolution (ICSME)*, pages 426–430, Washington, DC, USA, 2014. IEEE.

[41] L. Linsbauer, T. Berger, and P. Grünbacher. A classification of variation control systems. In *Proc. Int'l Conf. Generative Programming: Concepts & Experiences (GPCE)*, pages 49–62, New York, NY, USA, 2017. ACM.

[42] J. Loeliger and M. McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development.* O'Reilly Media, Inc., 2012.

[43] L. Luo. Synchronisierung von software-varianten mit variantsync. Master's thesis, University of Magdeburg, Germany, 2012. In German.

[44] Marlin Firmware Contributors. Marlin firmware. Website. Available online at `https://github.com/MarlinFirmware/Marlin`; visited on September 1, 2018.

[45] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake. *Mastering Software Variability with FeatureIDE.* Springer, Berlin, Heidelberg, 2017.

[46] M. Mendonça, A. Wąsowski, and K. Czarnecki. SAT-Based Analysis of Feature Models is Easy. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 231–240, Pittsburgh, PA, USA, 2009. Software Engineering Institute.

[47] W. Miller and E. W. Myers. A file comparison program. *Software: Practice and Experience*, 15(11):1025–1040, 1985.

[48] E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[49] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[50] J. A. Pereira, S. Schulze, S. Krieter, M. Ribeiro, and G. Saake. A Context-Aware Recommender System for Extended Software Product Line Configurations. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 97–104, New York, NY, USA, 2018. ACM.

[51] T. Pfofe, T. Thüm, S. Schulze, W. Fenske, and I. Schaefer. Synchronizing Software Variants with VariantSync. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 329–332, New York, NY, USA, Sept. 2016. ACM.

[52] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, Berlin, Heidelberg, Sept. 2005.

[53] D. Rabiser, P. Grünbacher, H. Prähofer, and F. Angerer. A Prototype-Based Approach for Managing Clones in Clone-and-Own Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 35–44, New York, NY, USA, 2016. ACM.

[54] D. Rabiser, P. Grünbacher, H. Prähofer, and F. Angerer. A prototype-based approach for managing clones in clone-and-own product lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 35–44, New York, NY, USA, 2016. ACM.

[55] L. Rincón, G. Giraldo, R. Mazo, and C. Salinesi. An Ontological Rule-Based Approach for Analyzing Dead and False Optional Features in Feature Models. *Electronic Notes in Theoretical Computer Science*, 302:111–132, 2014.

[56] J. Rubin, K. Czarnecki, and M. Chechik. Managing Cloned Variants: A Framework and Experience. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 101–110, New York, NY, USA, 2013. ACM.

[57] Sat4j Contributors. Sat4j - the boolean satisfaction and optimization library in java. Website. Available online at `http://www.sat4j.org/`; visited on September 1, 2018.

[58] K. Schmid, R. Rabiser, and P. Grünbacher. A Comparison of Decision Modeling Approaches in Product Lines. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 119–126, New York, NY, USA, 2011. ACM.

[59] K. Schmid and M. Verlage. The economic impact of product line adoption and evolution. *IEEE Software*, 19(4):50–57, Aug. 2002.

[60] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *Proc. Int'l Conf. Requirements Engineering (RE)*, pages 136–145, Washington, DC, USA, 2006. IEEE.

[61] C. Sontag. FeatFork : Feature Tracking In Preprocessor-Based Forks. Bachelor's thesis, Technische Universität Braunschweig, 2016.

[62] C. Sontag. VariantSync: Automating the Synchronization of Software Variants. Project work, Technische Universität Braunschweig, 2018.

[63] S. Stănciulescu, S. Schulze, and A. Wąsowski. Forked and Integrated Variants in an Open-Source Firmware Project. In *Proc. Int'l Conf. Software Maintenance and Evolution (ICSME)*, pages 151–160. IEEE, Sept. 2015.

[64] R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 81–86, New York, NY, USA, Oct. 2009. ACM.

[65] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming (SCP)*, 79(0):70–85, Jan. 2014.

[66] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. Abstract Features in Feature Modeling. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 191–200, Washington, DC, USA, Aug. 2011. IEEE.

[67] F. J. van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.* Springer, Berlin, Heidelberg, 2007.

[68] D. M. Weiss. The Product Line Hall of Fame. In *Proc. Int'l Software Product Line Conf. (SPLC)*, page 395, Washington, DC, USA, 2008. IEEE.

[69] D. M. Weiss and C. T. R. Lai. *Software Product-line Engineering: A Family-based Software Development Process.* Addison-Wesley, 1999.

[70] J. White, J. A. Galindo, T. Saxena, B. Dougherty, D. Benavides, and D. C. Schmidt. Evolving Feature Model Configurations in Software Product Lines. *J. Systems and Software (JSS)*, 87(0):119–136, 2014.

[71] K. Yoshimura, D. Ganesan, and D. Muthig. Assessing Merge Potential of Existing Engine Control Systems into a Product Line. In *Proc. Int'l Workshop Software Engineering for Automotive Systems (SEAS)*, pages 61–67, New York, NY, USA, 2006. ACM.

9 Appendix

Feature Model of the Marlin Firmware

The complete feature model for the Marlin firmware [44] which was extracted from *Marlin/Marlin/Configuration.h* can be found on the CD.

Evaluation

The evaluation results are also available on CD.

Pull Requests

The following pull requests were evaluated:
#7768, #7816, #7818, #7914, #7919, #7987, #7992, #8021, #8047, #8141, #8148

Configurations

The following configurations are assumed in the evaluation:

Variant 1 (Default)

- ULTRA_LCD
- EXTRA_FAN_SPEED
- 3 Fans,
- USB
- ! SD_SORT_CACHE_NAMES
- ! SDSORT_DYNAMIC_RAM
- TX_BUFFER_SIZE
- SCOLL_LONG_NAMES
- EEPROM_CHITCHAT
- HAS_TEMP_BED
- ! DISTINCT_E_FACTORS
- SHOW_TEMP_ADC_VALUES

Variant 2 (Creality CR-10S)

- STOCK DISPLAY
- 2 Fans
- Babystepping
- AUTO_BED_LEVELING_BILINEAR
- EEPROM_SETTINGS
- ADVANCED_OK
- EMERGENCY_PARSER
- SDSUPPORT
- EEPROM_SETTINGS
- ADVANCED_PAUSE_FEATURE
- SD_SORT_CACHE_NAMES
- ! SDSORT_DYNAMIC_RAM
- SDSORT_USES_RAM
- TX_BUFFER_SIZE
- SCOLL_LONG_NAMES
- EEPROM_CHITCHAT
- HAS_TEMP_BED
- ! DISTINCT_E_FACTORS
- SHOW_TEMP_ADC_VALUES

Variant 3 (Creality Ender 4)

- MINIPANEL
- 1 Fan
- 2 Serial Connections
- AUTO_BED_LEVELING_BILINEAR
- EEPROM_SETTINGS
- EMERGENCY_PARSER
- SDSUPPORT
- EEPROM_SETTINGS
- SD_SORT_CACHE_NAMES
- ! SDSORT_DYNAMIC_RAM
- TX_BUFFER_SIZE
- SCOLL_LONG_NAMES
- EEPROM_CHITCHAT
- HAS_TEMP_BED
- ! DISTINCT_E_FACTORS
- SHOW_TEMP_ADC_VALUES

Variant 4 (Scara)

- 3 Fans
- 2 Serials
- USB
- HAS_MOTOR_CURRENT_PWM
- ! SD_SORT_CACHE_NAMES
- ! SDSORT_DYNAMIC_RAM
- TX_BUFFER_SIZE
- SCOLL_LONG_NAMES
- EEPROM_CHITCHAT
- HAS_TEMP_BED
- ! DISTINCT_E_FACTORS
- SHOW_TEMP_ADC_VALUES

Variant 5 (Anet A8)

- 1602 display
- EXTRA_FAN_SPEED
- 2 Fans
- babystepping
- Z_DUAL_STEPPER
- EEPROM_SETTINGS
- SDSUPPORT
- EEPROM_SETTINGS
- ! SD_SORT_CACHE_NAMES
- ! SDSORT_DYNAMIC_RAM
- TX_BUFFER_SIZE
- SCOLL_LONG_NAMES
- EEPROM_CHITCHAT
- HAS_TEMP_BED
- ! DISTINCT_E_FACTORS
- SHOW_TEMP_ADC_VALUES

Variant 6 (OpenBeam Kossel Pro)

- EXTRA_FAN_SPEED
- 1 Fan
- 2 Serial Connections
- SDSUPPORT
- ! ENABLE_LEVELING_FADE_HEIGHT
- ! SD_SORT_CACHE_NAMES
- ! SDSORT_DYNAMIC_RAM
- TX_BUFFER_SIZE
- SCOLL_LONG_NAMES
- EEPROM_CHITCHAT
- HAS_TEMP_BED
- ! DISTINCT_E_FACTORS
- SHOW_TEMP_ADC_VALUES

Aufgabenstellung: Recording Feature Mappings During Evolution of Cloned Variants

Master Thesis - TU Braunschweig (DE)

Motivation

Software product lines (SPLs) are used by more and more companies [67, 68], as they have multiple advantages over clone-and-own when developing software variants [17, 56, 63]. On the other side, SPLs require the migration from cloned variant development to a software product line which can lead to a huge effort. As in general, the industry is mostly focused on a good payoff, they do not build up a complete new SPL in a proactive manner [3, 9, 16, 52]. They will build the first variant in a normal development process and also when another variant is requested clone-and-own still often have a better cost-benefit ratio then migrating them to a product line [14, 17, 59, 71]. Nevertheless, when reaching the payoff point [69] where enough variants exist, software product-lines gain a better cost-benefit ratio in the development process but have high migration costs as these are increasing for each variant and for each addition to the complexity of the software. Through these costs, many companies never switch to a product line as the costs are always too high, whether it is because of a worse cost-benefit ratio or high migration costs which significantly reduce the payoff [59].

Between the classic software development and software product lines, other research approaches have the goal to support clone-and-own, like computer-aided clone-and-own [36], ECCO [21], feature feeds [39], and the approaches by Rabiser et al. [53] and by Linsbauer et al. [40]. The development process is often nearly unchanged from the development process of the classic software development process, but with product line enhancements. Despite they often bring a better cost-benefit ratio for scenarios described above, they often lack a usable and practical feature mapping functionality as they often require interactions by the user during or after the development process.

Objective

Automated recording of changes and their mapping to feature contexts without user interactions could improve the usability and practicability of methods which attempt to enhance clone-and-own with software product lines. Despite that, automated recording of mappings leads to some problems like missing knowledge for mapping interactions as well as synchronization issues. Mapping and synchronizing feature contexts to their artifacts can sometimes open the question which feature context should be applied (e.g. the surroundings are already mapped, the line is already mapped, or the line is going to be removed). In these cases, composed logical formulas could enhance the feature context so that feature contexts can automatically be mapped. This thesis should create an approach for solving these issues in an conceptional framework manner with the possibilities to transfer the approach to different research ideas easily.

Tasks

1. Create a conceptional framework approach for feature context mappings. When artifacts are added, changed, or removed, the approach should provide a usable and practical way to conclude the right feature context with lightweight user interactions possible. Special focus should lay on interactions between feature contexts as they occur more and more during the evolution of variants and often need additional information which are often not provided when using automated feature context mapping.
2. Create a conceptional framework approach for synchronization of artifacts with feature context mappings. Mappings which add, change, or remove artifacts can conflict with mappings in target variants and should be resolved with usability and practicability in mind.
3. A prototypical implementation of the approaches in VariantSync [51]. To support the approaches during recording and synchronization, the line-based algorithm should be enhanced with a component that resolves interactions between feature context mappings. The Eclipse extension point for mapping algorithms should also be extended for future research approaches.
4. Evaluation of the approaches based on, for example, one of the following options:
   - A small self-made software product line which is similar to an industrial product line and uses VariantSync [51] from a certain point onwards.
   - A existing product line for which the history is available. This includes as well existing clones as also product lines created with different implementation techniques.
   - A experiment with students or an industrial partner which use VariantSync [51] to develop a product line variant.

   In all evaluation methods, the different cases of interactions between feature contexts as well as the synchronizations between variants should be collected and evaluated in the aspects of overhead to a classical development and their practicability.

https://doi.org/10.24355/dbbs.084-201812191127-0