TU Braunschweig

Master's Thesis

Semi-Automated Inference of Feature Traceability During Software Development

Author:

Paul Maximilian Bittner

January 30, 2020

Advisors:

Prof. Dr.-Ing. Ina Schaefer, Prof. Dr. rer. nat. Roland Meyer,
Prof. Dr.-Ing. Thomas Thüm, M.Sc. Tobias Pett,
Dr. Lukas Linsbauer, Prof. Dr. Timo Kehrer

https://doi.org/10.24355/dbbs.084-202002271120-0

https://doi.org/10.24355/dbbs.084-202002271120-0

Bittner, Paul Maximilian:
Semi-Automated Inference of Feature Traceability During Software Development
Master's Thesis, TU Braunschweig, 2020.

https://doi.org/10.24355/dbbs.084-202002271120-0

https://doi.org/10.24355/dbbs.084-202002271120-0

Abstract

Despite extensive research on software product lines in the last decades, ad-hoc clone-and-own development is still the dominant way for introducing variability to software systems. Therefore, the same issues for which software product lines were developed in the first place are still imminent in clone-and-own development: Fixing bugs consistently throughout clones and avoiding duplicate implementation effort is extremely difficult as similarities and differences between variants are unknown.

In order to remedy this, we enhance clone-and-own development with techniques from product-line engineering for targeted variant synchronisation such that domain knowledge can be integrated stepwise and without obligation. Contrary to retroactive feature mapping recovery (e.g., mining) techniques, we infer feature-to-code mappings directly during software development when concrete domain knowledge is present.

In this thesis, we focus on the first step towards targeted synchronisation between variants: the recording of feature mappings. By letting developers specify on which feature they are working on, we derive feature mappings directly during software development. We ensure syntactic validity of feature mappings and variant synchronisation by implementing *disciplined annotations* through abstract syntax trees. To bridge the mismatch between change classification in the implementation and abstract layer, we synthesise *semantic edits* on abstract syntax trees. We show that our derivation can be used to reproduce variability-related real-world code changes and compare it to the feature mapping derivation of the projectional variation control system VTS by Stănciulescu et al.

https://doi.org/10.24355/dbbs.084-202002271120-0

https://doi.org/10.24355/dbbs.084-202002271120-0

Inhaltsangabe

Trotz umfangreicher Forschung zu Software-Produktlinien in den letzten Jahrzehnten ist Clone-and-Own immer noch der dominierende Ansatz zur Einführung von Variabilität in Softwaresystemen. Daher stehen bei Clone-and-Own immer noch die gleichen Probleme im Vordergrund, für die Software-Produktlinien überhaupt erst entwickelt wurden: Die konsistente Behebung von Fehlern in allen Klonen und die Vermeidung von doppeltem Implementierungsaufwand sind äußerst schwierig, da Ähnlichkeiten und Unterschiede zwischen den Varianten unbekannt sind.

Um hier Abhilfe zu schaffen, erweitern wir die Clone-and-Own-Entwicklung mit Techniken aus der Produktlinien-Entwicklung zur gezielten Synchronisierung von Varianten, sodass Entwickler ihr Domänenwissen schrittweise und unverbindlich integrieren können. Im Gegensatz zu nachträglich arbeitenden Feature-Mapping-Recovery- oder auch Mining-Techniken, leiten wir Zuordungen von Features zu Quellcode direkt während der Softwareentwicklung ab, wenn konkretes Domänenwissen vorhanden ist.

In dieser Arbeit entwickeln wir den ersten Schritt zur gezielten Synchronisation von Varianten: die Aufzeichnung von Feature-Mappings. Indem Entwickler spezifizieren an welchem Feature sie arbeiten, leiten wir Feature-Mappings direkt während der Softwareentwicklung ab. Wir stellen die syntaktische Korrektheit von Feature-Mappings und der Synchronisation von Varianten sicher, indem wir disziplinierte Annotationen mithilfe von abstrakten Syntaxbäumen implementieren. Um die Diskrepanz der Klassifizierung von Änderungen zwischen der Implementierungs- und der Abstraktionsschicht zu überbrücken, synthetisieren wir *Semantic Edits* auf abstrakten Syntaxbäumen. Wir zeigen, dass unsere Ableitung von Feature-Mappings in der Lage ist reale Codeänderungen zu reproduzieren und vergleichen sie mit der Feature-Mapping-Ableitung des Variationskontrollsystems VTS von Stănciulescu et al.

https://doi.org/10.24355/dbbs.084-202002271120-0

https://doi.org/10.24355/dbbs.084-202002271120-0

Contents

https://doi.org/10.24355/dbbs.084-202002271120-0

https://doi.org/10.24355/dbbs.084-202002271120-0

List of Figures

https://doi.org/10.24355/dbbs.084-202002271120-0

https://doi.org/10.24355/dbbs.084-202002271120-0

List of Tables

https://doi.org/10.24355/dbbs.084-202002271120-0

https://doi.org/10.24355/dbbs.084-202002271120-0

List of Code Listings

https://doi.org/10.24355/dbbs.084-202002271120-0

https://doi.org/10.24355/dbbs.084-202002271120-0

1. Introduction

Modern software is often required in form of multiple variants. Naturally, software development usually starts with just a single variant to reduce complexity and costs or because the need for future variants is commonly unknown [AJB+14, LFLHE15]. When the demand for a new variant emerges, one approach is cloning the whole software to alter specific parts independently from the previous variant. This ad-hoc solution is known as clone-and-own [AJB+14, DRB+13, RCC13, SSW15]. However, propagating changes in clone-and-own development such as bug fixes to other clones is increasingly difficult and ambiguous for a growing number of software clones. Developers are usually only familiar with a subset of variants, so it remains unclear which variants are possible targets for change synchronisation. For the same reason, it is not obvious in which variants a certain feature is already implemented when required in a source variant. Thus, loss of software quality, duplicate implementation effort, and higher maintenance costs are imminent.

Software product lines allow managing variants by mapping implementation artefacts to features. Through careful domain engineering, common features between planned variants are identified in advance. Dedicated generation mechanisms compose feature implementations to target variants of any configuration. That way, features can be shared and reused across variants [ABKS13, CE00]. Although product lines are a dedicated mechanism for reusing software artefacts between different software products, they are rarely adopted in practice due to uncertainty on desired variants at the beginning of development, a high up-front investment, lack of tool support, and necessary workflow adaptations [DRB+13, RCC13].

As clone-and-own is still the favoured approach for introducing software variability [DRB+13, RCC13], lots of research focuses on migrating clone-and-own software to product lines [KDO14, FMS+17, KFBA09, LC13, WSSS16]. Most existing techniques rely on elaborated heuristics to retroactively recover feature mappings, thus suffering from uncertainty due to potential loss of domain knowledge. Usually, such migrations require to halt development for an unknown time span. What is more, lots of legacy applications using the clone-and-own approach bore so many variants over decades that a migration to a software product line is not only time-intensive

https://doi.org/10.24355/dbbs.084-202002271120-0

and challenging but also not guaranteed to succeed. Feature recovery tools require numerous developer decisions [FMS+17, FLLHE15, KDO14, KKK13, LLHE17, MZB+15, RCC13, ZHP+14] and fully automatic migration techniques [FLLHE15, LLHE17, WSSS16, ZHP+14] suffer from *unintentional divergence* [KKK13, SL14]. Moreover, the necessary domain knowledge for product-line migration is distributed widely because in clone-and-own development each developer is usually responsible for a single variant only [AJB+14, DRB+13, LnBC16, RCC13, SSW15].

To address the risks of migration to software product lines, we target the gradual introduction of domain knowledge at will without impairing ongoing software development. To this end, we aim at synchronising changes between clone-and-own variants through software product-line technology. Possible target variants for changes can be identified automatically when it is known to which features an implementation artefact belongs and which variants implement those features. Therefore, we introduce the product-line engineering concepts of features and configurations to clone-and-own development. As our concepts are not strongly tied to the clone-and-own scenario or any concrete programming language, our insights on recording feature mappings are also useful for software product-line engineering in general.

In this thesis, we develop the very first step towards feature-driven targeted synchronisation between software clones: recording of feature mappings. As feature mappings are a basic requirement for the automated synchronisation between variants, they have to be introduced beforehand. To minimise the effort of specifying feature mappings, we derive them semi-automatically from implementation artefact changes. We employ *disciplined annotations* [KAT+09] to ensure syntactical correct feature mappings, such that removing a feature does not introduce syntax errors. Therefore, we use an abstract representation of implementation artefacts, so called Abstract Syntax Trees (ASTs).

We extend existing work on AST-based annotations [KAK08] by introducing semantic edits. Developer's changes to the external representation of implementation artefacts (i.e. in a text editor) are more coarse-grained and more intricate than sole node-based ASTs operations. For instance, surrounding existing statements with a newly inserted condition corresponds to a subtree insertion with attendant tree restructuring. However, single-node and technical tree operations are widely used in the literature for tree diff computation [Bil05, PA11, FMB+14, CRGMW96, HM08]. We elaborate on how a technique known as *semantic lifting* [KKT11] can be adopted to recover user-level semantic edits from fine-grained technical edit scripts computed by existing tree differencing algorithms [PA11, FMB+14, CRGMW96, HM08].

We derive feature mappings upon software changes by incorporating developers' knowledge on which feature they are currently implementing. As interactions of features are common in variability enabled software development, developers specify a propositional formula over the set of features, the so called *feature context*. During development, edited software artefacts are assigned to the currently active feature context. However, it is yet unclear how to deal with already existing mappings or deletions of artefacts. For instance, it is unknown if a method inserted into a class with mapping A under feature context B should be assigned to A, B, $A \wedge B$, or any other expression. Existing work on this topic [Son18] did not consider the structure

https://doi.org/10.24355/dbbs.084-202002271120-0

of implementation artefacts and thus had to rely on unreasonable assumptions about dependencies between source code lines.

We present an algorithm for deriving feature mappings upon insertions, deletions, moves, and updates of software artefacts in a reasonable, consistent, and distinct way. As our main goal is the gradual synchronisation of implementation artefacts across variants, our derivation resolves ambiguities and inconsistencies stepwise, i.e., each time an edit under a specific feature context is made.

The main contributions of this thesis can be summarised as follows:

Semantic Edits on ASTs – Intuitively classified changes of implementation artefacts do not need to correspond to alike changes in the corresponding AST. Therefore, we identify a subset of operations on ASTs corresponding to reasonable (i.e. semantic) edits in the implementation layer.

Consideration of Semantic Lifting on AST Edit Scripts – To express AST diffs as a series of semantic edits, we elaborate on how semantic lifting can allow reusing existing tree diffing algorithms.

Semi-Automatic Feature Mapping Derivation – We present an algorithm for feature mapping derivation from coarse-specified artefact changes by incorporating the knowledge on which feature or feature interaction developers are currently working on. Thereby, we especially take care of reasonably treating existing mappings. We show how the derived feature mappings identify target variants for change synchronisation.

Null Feature Mappings – Opposed to previous concepts [SBWW16], we specifically consider the absence of a user-specified feature context. We support cases in which developers do not have the required domain knowledge and thereby do not enforce a strict, immediate workflow adaption. Our derivation does not require a set feature context and preserves existing feature mappings.

Evaluation of Applicability – We evaluate our feature mapping derivation in the clone-and-own scenario by reproducing real-world variability-related code changes identified by Stănciulescu et al. [SBWW16]. Thereby, we show that our derivation can be used to reenact all variability-related code changes in the history of the printer firmware Marlin [vdZ]. We compare it to the projectional software product-line variation control system VTS [SBWW16]. Our derivation requires slightly simpler user-interactions while being as powerful in terms of specifying feature mappings.

We start by introducing the concepts of clone-and-own and software product lines as well as notations we use throughout this thesis in Chapter 2. In Chapter 3, we discuss the advantages and disadvantages of AST-based feature mappings compared to the straightforward *line*-based mappings, and formalise disciplined annotations on ASTs. In Section 3.3, we introduce semantic edits which ensure that edit operations in the AST correspond to reasonable changes in the text-based representation. Further, we elaborate on how semantic lifting enables reusing existing tree-diffing

https://doi.org/10.24355/dbbs.084-202002271120-0

algorithms to compute tree diffs of semantic edits. In Chapter 4, we develop our semi-automatic feature mapping derivation during software development. By considering four types of edits, we show how our derived feature mapping identifies variants as targets for change synchronisation. We address four important technical challenges that need to be faced upon implementing our derivation in Chapter 5, by discussing possible solutions. We evaluate the applicability of our feature mapping derivation for targeted synchronisation of clone-and-own variants in Chapter 6. In Chapter 7, we discuss existing work on this topic, how we use it and how it can be used in the future. We summarise the results of this thesis in Chapter 8. Finally, we give an outlook on potential further research topics in Chapter 9.

https://doi.org/10.24355/dbbs.084-202002271120-0

2. Background

In this chapter, we summarise background knowledge relevant for this thesis. Each concept is explained in its own section. We begin with explaining the concept of software product lines and thereby the meaning of features and configurations, which we use extensively throughout this thesis, in Section 2.1. In Section 2.2, we explain the widely used clone-and-own approach of variability management in software development which we aim to improve. To classify the level of adaption of product-line techniques we refer to the scheme introduced by Antkiewicz et al. [AJB+14] described in Section 2.2.1. In Section 2.2.2, we cover existing approaches for migrating clone-and-own software to product lines. We introduce notations used in this thesis in the end in Section 2.3.

2.1 Software Product Lines

Large scale software that is available in multiple variants is referred to as a software product line because it comprises different products in a single code base. In contrast to usual single system software engineering, product-line engineering is divided into two phases: *domain* and *application engineering* [ABKS13, PBvdL05]. Figure 2.1 illustrates both phases. In the initiating *domain engineering* phase, the target domain is analysed for commonalities and differences. The software requirements are analysed to distil features which describe individual characteristics visible to the user. Different combinations of these features (configurations) are designated to form the final products or variants [ABKS13]. Therefore, features are mapped to implementation artefacts such that each implementation artefact corresponds to a feature or a feature interaction. By installing dedicated variation mechanisms, features can be composed to variants or products [ABKS13, CE00]. For instance, the C/C++ preprocessor is a well known and widely used variation mechanism as it allows in- or excluding source code lines trough special statements (`#ifdef`) during compilation depending on pre-defined flags, e.g., the configuration. *Application engineering* describes the process of selecting a set of features and deriving its final software product either manually or fully automated. In preprocessor-based product lines, this

https://doi.org/10.24355/dbbs.084-202002271120-0

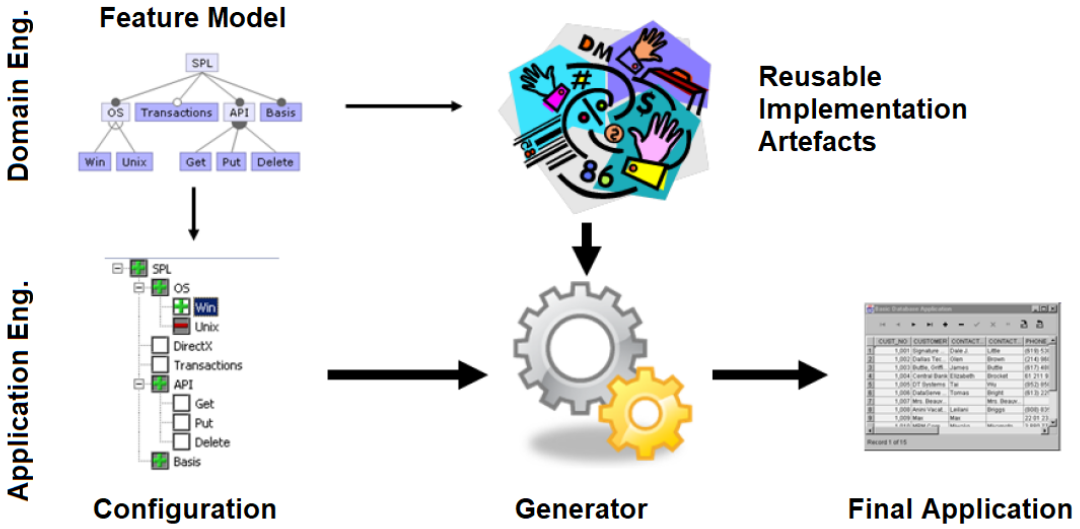


Figure 2.1: Schematic Structure of Software Product-Line Engineering (Adapted From [Thü18])

comprises the compilation of the code base with features `F` in the target configuration being defined as preprocessor statements / flags, e.g., trough `#define F`. In the following, we explain the concepts of feature models and feature mappings. Feature models describe valid configurations of features. Such that the product generator is able to compose implementation artefacts, it has to know which implementation artefacts belong to which features, which is described by feature mappings.

2.1.1 Feature Models

In software product lines individual features are composed to variants or products. Thus, any combination of features results in a different variant. However, not all combinations may be desired or are technically realisable. For instance, platform independent software may have to initialise different routines depending on the operating system it runs on. As the operating system does not change during runtime, such variability checks could be performed pre-emptively to completely exclude unused software artefacts. Therefore, feature models are used to not only specify the features of a software product line but their valid combinations, known as configurations, by embedding the features in a tree hierarchy [Bat05, CE00]. Features can only be selected in a configuration if their parent features are also selected. That way dependencies between features can be modelled. Furthermore, child features can be grouped in special group types, such as alternatives, in whose exactly one child feature has to be selected, or disjunctions, where at least one child has to be selected. If certain constraints on configurations cannot be expressed with the tree hierarchy only, cross-tree constraints can be specified manually. Usually, cross-tree constraints are specified as a propositional formula over the set of features as the feature model itself is converted to a propositional formula for analysis purposes [Bat05]. The set of all possible combinations of features described by a feature model are the configurations. Thereby, each configuration corresponds to exactly one unique software variant as configurations are pairwise disjunct.

https://doi.org/10.24355/dbbs.084-202002271120-0

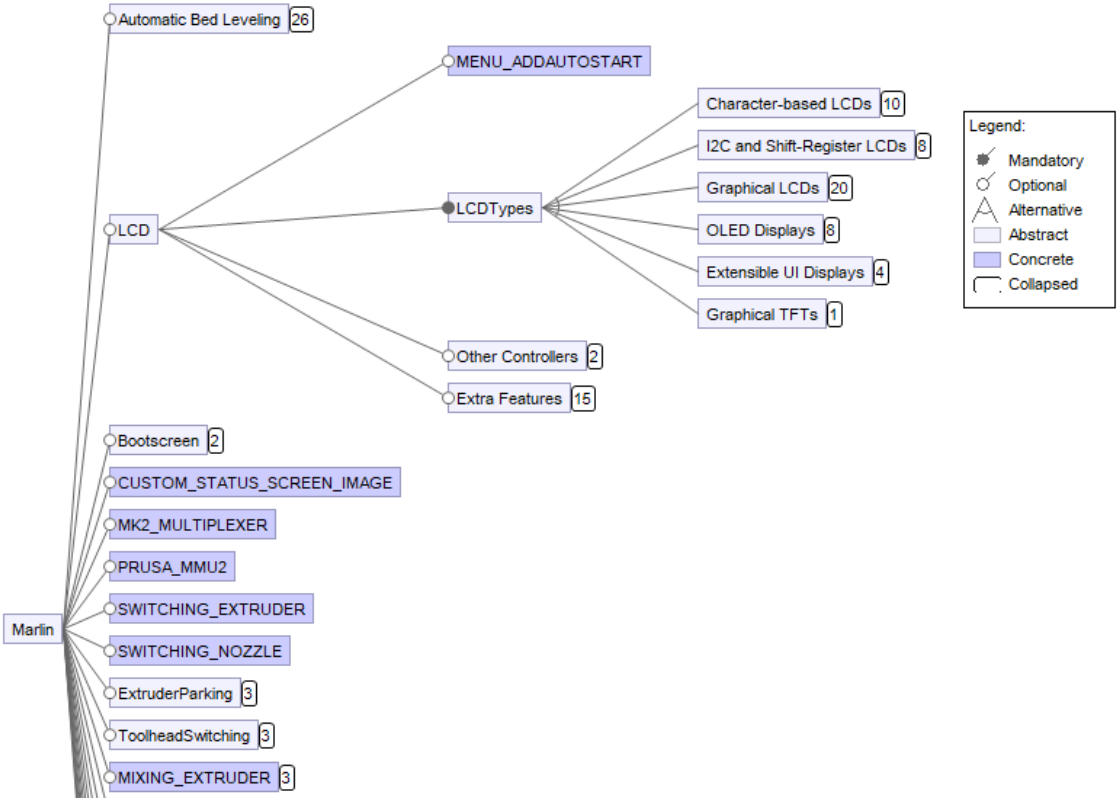


Figure 2.2: Excerpt of the Feature Model of the Marlin Firmware

Example 2.1.1. Figure 2.2 shows an excerpt of the feature model of the Marlin firmware [vdZ]. Numbers behind features indicate the number of child features being collapsed for a more clear overview. The feature `LCD` is abstract as it does not occur in the implementation, i.e., no software artefacts are mapped to it. Nevertheless, as each LCD screen has a specific type which has to be known for its runtime setup, the feature `LCDTypes` has to be chosen, when an LCD screen is present, i.e., feature `LCD` is present. Thus, feature `LCDTypes` is mandatory. As the LCD screen has one but not more than one type, exactly one actual type has to be chosen, thus forming an *alternative* group. Again, the concrete LCD screen types are grouped by categories `Character-based LCDs`, `I2C and Shift-Register LCDs`, `Graphical LCDs`, for convenience but without any implementation effects.

Some software artefacts may be part of every software variant. Such features are referred to as *core features*. Features are core features if they are a mandatory feature below the root feature or if they are a mandatory child of another core feature. Complementary, dead features are those features that cannot be part of any variant. Features being dead is usually an indication for ill-formed constraints as dead features to do not contribute to any variant. If dead features exhibit actual implementation artefacts, those artefacts do not serve any purpose for variant generation.

https://doi.org/10.24355/dbbs.084-202002271120-0

```
1  #if HAS_GAMES
2
3  #include "game.h"
4
5  int MarlinGame::score;
6  uint8_t MarlinGame::game_state;
7  millis_t MarlinGame::next_frame;
8
9  MarlinGameData marlin_game_data;
10
11 bool MarlinGame::game_frame() {
12 // code for rendering a game's frame
13 }
14
15 #endif // HAS_GAMES
```

Listing 2.1: Example for Preprocessor Statements Used in Marlin Firmware

2.1.2 Feature Mappings

For assigning features to implementation artefacts, two distinct approaches are in use: the *annotative* and the *compositional* approach [KAK08]. Depending on the shape of feature mappings, different generators are necessary.

In the annotative approach, features are usually implemented in a single code base and annotated with their corresponding feature or feature interaction. Annotations can either be specified internally or externally [KAK08]. Internal annotations are located directly inside the implementation artefact. A common mechanism for internal annotations are the `#if`, `#ifdef`, and `#endif` statements of the C/C++ preprocessor. Common software product lines using preprocessor annotations are the Linux Kernel [Tor] and the Marlin Firmware [vdZ] for 3D printers. Both use the C preprocessor, a built-in language in C/C++, to conditionally include or exclude certain parts of the source code during compilation. As an example, Listing 2.1 shows an excerpt of an implementation for games inside Marlin that can be played on the 3D printers screen. The game code is only included to the software if the preprocessor macro `HAS_GAMES` is set to value other than 0. Thus, depending on the printers specifications, games can either be en- or disabled. External annotations are specified in the representation layer and stored aside of the actual implementation artefact. For example, in the Colored Integrated Development Environment (CIDE) by Kästner et al. [KAK08] features are assigned colours as shown in Figure 2.3, only visible in dedicated editors for source code.

In compositional approaches, features are implemented as distinct modules that extend a common base software [KAK08]. To compose those features, dedicated software architectures like frameworks [JF88], component technologies [SGM02], or specialised languages like aspects [KLM+97], or feature-oriented programming [BSR04] are required. Next to dedicated plug-in based frameworks, dedicated research focuses on other modular ways for feature specification and composition. In *feature*

https://doi.org/10.24355/dbbs.084-202002271120-0

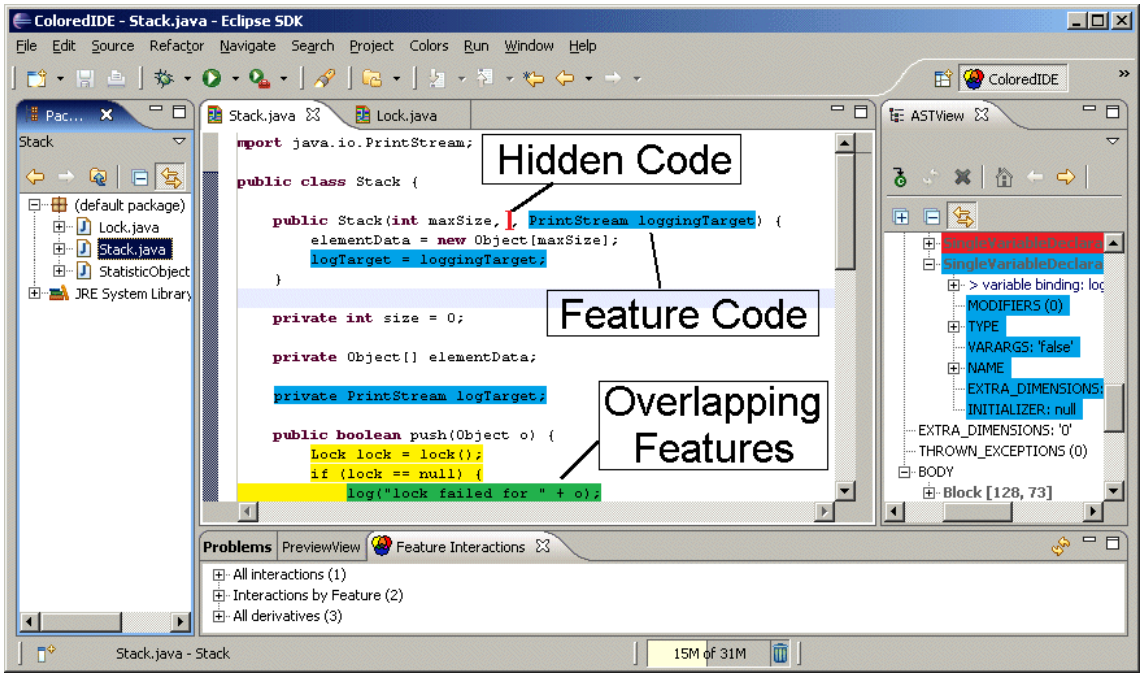


Figure 2.3: Example of External Feature Mappings Specified in the Colored Integrated Development Environment [KAK08]

oriented programming [Pre97], roles are the primary development artefact. Roles are parts of classes implementing exactly one feature in that class. By composing different roles according to the configurations, the final class is generated, implementing said configuration. Thus, each feature is implemented in several roles as there is possibly more than one class implementing a feature. In *aspect oriented programming* [KLM+97], a core software is extended by so called *aspects*. The core software is exactly that part of the entire software being present in every variant (i.e. belonging to core features). Aspects refine the core software with externally specified code by locating areas to extend through regular expressions, so called *pointcuts*. To obtain the final variant, the aspects corresponding to the features selected in the configuration are *weaved* into the core software.

2.2 Clone-and-Own

Managing software variability in a large scale is yet difficult and often requires custom solutions. Different approaches to describe the commonly orthogonal differences between software variants have shown useful in research and industrial projects. The straightforward approach of copying the entire source code to alter specific parts is known as clone-and-own and allows arbitrarily fine-grained adjustments. The overhead of employing dedicated variability managing solutions, such as conditional compilation, component- or plugin-based architectures, is omitted by introducing a source code clone for each variant. As no initial costs arise and no prior knowledge and planning are necessary, clone-and-own is willingly used if it is unknown which and how many variants will be needed in the future [DRB+13, RCC13].

However, for larger amounts of variants, maintaining updates, such as bug-fixes, between them becomes increasingly complicated and tedious as it is often unknown

https://doi.org/10.24355/dbbs.084-202002271120-0

which clones should be affected by changes. For the same reason it is often unclear to developers that certain features they implement may already exist in other clones resulting in duplicate effort and costs. Furthermore, with an increasing number of clones, migrating the software to a more controlled management environment becomes increasingly challenging, time-consuming, and thus more expensive. As the introduction of just a single variant does not justify such a migration of the whole software, it gets postponed usually.

2.2.1 Virtual Platform

Antkiewicz et al. [AJB+14] introduced a classification scheme to describe the continuum between ad-hoc clone-and-own (L0), where the different software variants are first-class development artefacts, and pure product-line engineering (L6), where the software's features are target for development instead of the final products. The classification consists of seven levels L0 - L6 in total. Each level adds another variability management mechanism to the previous level.

Level L0 comprises pure clone-and-own software development. In that sense, no domain knowledge is made explicitly. Thus, it is unknown how variants were cloned and how they evolved. Level L1 is known as *Clone-and-Own with Provenance* and keeps track of variant changes and points in time when new clones emerge. Usually, *version control systems* are used to record the cloning process as they explicitly allow cloning, known as branching and forking, and enable fast navigation between those branches. Furthermore, version control systems, such as *git*, *SVN*, *Mercurial*, *Subversion*, and *Perforce Helix* are in wide use for single system engineering [CW98, Tic82] as they not only allow clone-and-own development but team collaboration and to back up the source code, too. Due to their wide acceptance and ad-hoc applicability, the hurdle for introducing those tools to development in the first place is marginal. Level L2 is described as *Clone-and-Own with Features* where features are identified as development artefacts across the clones. It is the first step to exploiting domain knowledge for clone synchronisation as features are synchronised between clones. To further reduce the effort for cloning, Level L3 *Clone-and-Own with Configuration* identifies the features implemented in a variant by assigning a configuration to each variant. Each clone itself may contain variability mechanisms for deselecting features it contains. Thereby, multiple variants can be cloned by choosing a subset of the features of an existing clone. With Level L4 *Clone-and-Own with a Feature Model*, a single global feature model is introduced to describe the variability across all clones. It contains all implemented features as well as global constraints of them leading to the existing clones. Level L5, *Product-Line Engineering with an Integrated Platform and Clone-and-Own* describes actual software product-line engineering but application engineering is not automated. A manual post-processing is necessary to compose the features to a final product. The final level, L6 *Product-Line Engineering with a Fully Integrated Platform*, fully automates software product-line engineering as explained in Section 2.1.

We refer to this scheme throughout the thesis for context clarification.

2.2.2 Migration to Software Product Lines

To open product-line development to existing software, many research projects focus on extracting features from existing code [KDO14, FMS+17, KFBA09, LC13,

https://doi.org/10.24355/dbbs.084-202002271120-0

WSSS16]. Generally, there are three strategies for implementing a software product line [ABKS13, Kru02, PBvdL05]:

Proactive – Implementing software product lines from scratch without any prior artefacts is referred to as the proactive strategy. It can be used when (nearly) no existing implementation is present. Therefore, extensive expertise and a complete analysis of the domain are necessary.

Reactive – The reactive strategy can be seen as a soft and less risky version of the proactive strategy and is analogous to extreme programming. Development starts with a small amount of variants to which incrementally new variants are added over time. The reactive strategy is useful when it is not clear at beginning of development which and how many variants are needed. Though, later refactorings of the product line may be necessary to adapt to unexpected changes.

Extractive – Migrating existing software variants to a single software product line is known as the extractive strategy as variability information is extracted retroactively. This strategy is useful for already existing traditional software projects but very challenging for tools and developers as the whole software has to be restructured.

The most commonly used strategy is the extractive strategy [BRN+13]. Migrating existing software to a product line consists of two major phases [MZB+15]. First, variability information needs to be extracted. This step is known as variability mining or family mining [KDO14, WSSS16] and focuses on recovering feature mappings as well as synthesising a feature model [MAaL18] describing the detected variability. Second, features need to be extracted and composed to the final product line with a convenient mechanism [FMS+17, KFBA09, LC13]. This is done either seamless, also referred to as *big bang migration*, or stepwise [FMS+17].

We target to enhance clone-and-own development and not to migrate it to a software product line. Nevertheless, the recording of feature mappings and thereby stepwise synchronisation of implementation artefact between clones can be used for a later migration, if so wished.

2.3 Notation

In this section, we introduce names and notations we use along this thesis. Let $\mathcal{P}(X)$ denote the power set of a set X.

Definition 2.1 (Propositional Formula Space)**.** *Let the set of all formulas in propositional calculus over a given set of variables X be denoted as $\mathcal{B}(X)$. Let $\mathcal{B} := \bigcup_X \mathcal{B}(X)$ denote the set of all propositional formulas over all sets of variables X. Note that $\{\text{false}, \text{true}\} \subset \mathcal{B}(\emptyset) \subset \mathcal{B}(X) \subset \mathcal{B}$ for any set of variables $X \neq \emptyset$.*

An assignment to propositional formulas $\mathcal{B}(X)$ over variables X is a subset of the variables $A \subseteq X$, containing exactly those variables that are assigned to *true*,

https://doi.org/10.24355/dbbs.084-202002271120-0

whereas all missing variables $X \setminus A$ are assigned to *false*. If an assignment A satisfies a formula φ, i.e., φ evaluates to *true* under A, we write $eval(A, \varphi)$, which itself is a propositional predicate and thereby a propositional formula. Hence, we deliberately use it as such.

The set of features in the current feature model is referred to as F. Feature models are explained in detail in Section 2.1.1 on Page 6. As we assign features to previously unmapped implementation artefacts gradually, we also need to support an undefined feature mapping. Hence, we introduce *null* as a possible value for specific propositional variables. If *null* is a possible value, we explicitly mention it. In ternary logic, *null* is interpreted as an *unknown* or *maybe* state. However, in our case *null* does not represent an unknown state but the absence of a feature mapping. Therefore, we define it to be the neutral element for all operations in propositional calculus, i.e., $\varphi \wedge null :\equiv \varphi$, $\varphi \vee null :\equiv \varphi$, $\neg null :\equiv null$, $\varphi \Rightarrow null :\equiv true$, $null \Rightarrow \varphi :\equiv false$, and $eval(A, null) := true$ for $\varphi \in \mathcal{B}(X)$, arbitrary set of variables X, and an assignment A. Nevertheless, we explicitly mention possible *null* values and handle these cases with care, e.g., by checking for its presence.

https://doi.org/10.24355/dbbs.084-202002271120-0

3. Semantic Edits on Abstract Syntax Trees

In this chapter, we show how to represent feature mappings in a syntax preserving way. Therefore, we use an abstract representation of implementation artefacts to obtain knowledge on their hierarchical structure. To distinguish the types of changes developers make, we show how to compute differences between artefact versions on the abstract representation layer. Using semantic lifting on those differences, we are able to reflect the developer's intentions on changes more accurately.

Intuitively classified changes on implementation artefacts do not need to correspond to alike changes in the abstract representation. Depending on its content, inserting a line of source code can result in various changes in the Abstract Syntax Tree (AST), that we use as abstract representation for feature mappings. Therefore, we introduce definitions for *semantic edits*, i.e., operations on ASTs representing intuitive operations in their corresponding implementation artefact. However, existing tree differencing algorithms use technical, tree-oriented (i.e., low-level) operations to express tree diffs. Thus, classifying changes in the abstract representation the same way as developers would intuitively classify them in the implementation layer is still an open topic. *Semantic lifting*, a technique know from model-driven software development [KKT11], can be used to detect semantic edits in series of low-level edits. It has not been applied to AST diffs yet.

In the following, we first establish our assumptions on the clone-and-own scenario in Section 3.1 and how product-line techniques are integrated in concrete. Second, we discuss on how to represent feature mappings in Section 3.2. Third, we elaborate on how changes in implementation artefacts play out in their abstract representation in Section 3.3. Thereby, we introduce *semantic edits* on ASTs and how existing tree differencing algorithms could be used to derive semantic tree diffs, i.e., diffs of semantic edits. Finally, we summarise this chapter in Section 3.4.

https://doi.org/10.24355/dbbs.084-202002271120-0

3.1 Development Setting

To enrich clone-and-own development with product-line techniques, we make the following assumptions:

1. Software clones originate from each other, e.g., by branching in a version control system. Thus, they all have commonalities and deliberately introduced differences. This corresponds to the usual definition of clone-and-own [AJB+14, DRB+13, RCC13, SSW15]. Though, we do not require knowledge on how clones emerged from each other.

2. Developers agree on a common domain of features they target to implement or have already implemented. Such domain knowledge is commonly already given in documentations [RC13] or other departments, such as the sales department [LLHE17]. Optionally, a single global feature model is given to describe constraints on the features shared across all variants. It is not required but enables us to further reason about valid feature mappings. In the following, we do not need to distinguish between cases with and without a feature model as we can construct a basic feature model just containing the features without any constraints. Thus, if no feature model is given, whenever we refer to the global feature model from now on, we refer to the constraintless model, where all features are optional children under a single abstract base feature.

3. It is known which features are implemented in which variant, i.e., configurations are given for each of them, as assumed by others [FLLHE15, LFLHE15, LLHE17]. These configurations conform to the optional global feature model to ensure valid implementations. We assume these configurations to be constant because a change in a configuration can be achieved by introducing a new clone. Furthermore, from a technical point of view, as variants are supposed to differ but also have commonalities as they belong to the same software, those differences and commonalities can be identified as features that may or may not be implemented in a certain variant. Differing implementations of features between variants can be expressed by introducing sub-features describing the differences or could be identified as feature interactions.

4. Usually, developers know on which feature or feature interaction they are working on. As developers need an intention on what to achieve when programming and software teams often use some sort of task documentation, such as issue trackers or ticketing systems, we assess this assumption to be reasonable, as in other research [JBAC15, KDO14, DRB+13].

Figure 3.1 gives an overview on our assumptions by summarising the clone-and-own scenario we target. Features are illustrated by coloured boxes. Each variant is identified by the set of features it implements, depicted by a unique configuration. The set of available features is denoted by a single global feature model. Optionally, it can be enriched by constraints on feature composition. Here, the orange feature is a core feature because it is the root in the feature model. Hence, it is contained in every variant.

https://doi.org/10.24355/dbbs.084-202002271120-0

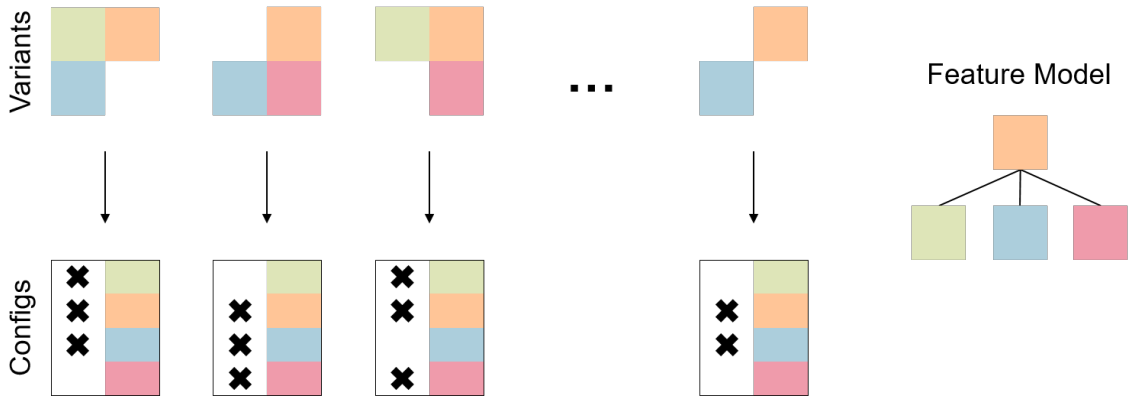


Figure 3.1: Overview on Feature Enhanced Clone-and-Own Development Scenario

Our assumptions are general enough to target multiple levels of clone-and-own development and any inclusion degree towards product-line engineering. Formally, we target clone-and-own development starting from level L2 up to level L4 if a feature model is given according to the scheme by Antkiewicz et al. [AJB+14], described in Section 2.2.1. Clone-and-own development where features are identified as development artefacts is referred to as L2.

Version control systems, such as *git*, *Mercurial*, or *SVN* are not a requirement for us although we have to reason about software changes, known as *diffs*. Our concept is largely orthogonal to version control systems, thus supporting any of them used currently for software development. Version control systems could ease the implementation of targeted synchronisation between variants but are out of scope of this thesis.

3.2 Feature Mapping Representation

Feature mappings, also known as feature traces, are mappings from features to their implementation artefacts, such as source code. They identify artefacts as the implementation of a certain feature. By composing those artefacts, different program variants are obtained. Thus, mapped artefacts must be removable from the overall program's implementation without invalidating the program. Dependencies between features can be expressed through combinations of them, so called *feature interactions*, or explicit constraints given in a feature model. In the following, we explain how we represent feature mappings and why.

Software artefacts cannot only belong to single features but feature interactions or even negations of features. As an example, consider Listing 3.1. If both features `FAN_SOFT_PWM` and `FAN_MIN_PWM` are active, the speed of the printers fan `fanSpeedSoftPwm` is calculated differently than in the absence of feature `FAN_MIN_PWM`. Consequently, lines 3 to 6 are mapped to both features described by the formula

$$\texttt{FAN\_SOFT\_PWM} \wedge \texttt{FAN\_MIN\_PWM}.$$

Accordingly, Line 8 is mapped to

$$\texttt{FAN\_SOFT\_PWM} \wedge \neg\texttt{FAN\_MIN\_PWM}.$$

https://doi.org/10.24355/dbbs.084-202002271120-0

```
1  #ifdef FAN_SOFT_PWM
2    #ifdef FAN_MIN_PWM
3      fanSpeedSoftPwm = (tail_fan_speed == 0) ?
4          0 : ((tail_fan_speed < FAN_MIN_PWM) ?
5            FAN_MIN_PWM
6            : tail_fan_speed);
7    #else
8      fanSpeedSoftPwm = tail_fan_speed;
9    #endif // FAN_MIN_PWM
10 #else
11 // other cases
```

Listing 3.1: Example for Feature Interactions and Negative Feature Mappings from an Old Version of the Marlin Firmware

Therefore, we allow arbitrary propositional formulas over features as feature mappings. From now on, if we use the term feature mapping, we refer to such formulas instead of single features. Notice that feature `FAN_MIN_PWM` is a numerical feature in Listing 3.1 as its value is used in lines 5 and 6. For now, we only support boolean features. Some usages of numerical features, such as presence checks (Line 3), can be reduced to boolean features, though.

As we want to trace features in software clones but not extract them, we use the annotative approach for feature mapping specification. Software artefacts can be annotated either *internally* or *externally*. Internal annotations are specified in the implementation layer, e.g., with preprocessor macros. Contrary, external annotations are specified in the representation layer (i.e., the file editor) and stored externally. They can be visualised by assigning colours to them, such as in CIDE [KAK08]. As we do not map features to implementation artefacts directly but to an abstract representation of them, as explained in Section 3.2.1, we have to specify mappings externally.

3.2.1 Abstract Syntax Trees as Feature Mapping Targets

Inspired by preprocessor annotations, the straightforward approach for specifying feature mappings in text files, such as source code, is the *line-based* mapping. Implementation artefacts are referenced by storing offset and length of their occurrence inside the documents containing them. Although this method is transparent and intuitive, it has several issues, especially regarding implementation artefacts exhibiting syntax such as source code. First, line-based mappings are not stable against changes. Changing a document outside of an appropriate development environment likely invalidates all mappings behind the location changes were made. Second, syntax-violating annotations are possible as shown in Listing 3.2. In Line 7, a closing bracket is wrongly mapped to ¬`AUTO_FILAMENT_CHANGE` (highlighted in red) leading to a syntactically invalid program for variants where feature `AUTO_FILAMENT_CHANGE` is not present. Third, and most important, line-based mappings allow no reasoning about the structure of implementation artefacts. Detecting dependencies between artefacts is essential for a sophisticated derivation of feature mappings.

https://doi.org/10.24355/dbbs.084-202002271120-0

```
1  while (!lcd_clicked()) {
2    #ifndef AUTO_FILAMENT_CHANGE
3      if (++cnt == 0) lcd_quick_feedback();
4      manage_heater();
5      manage_inactivity(true);
6      lcd_update();
7    }
8    #else
9      current_position[E_AXIS]+=/* [...] */;
10     plan_buffer_line(/* [...] */);
11     st_synchronize();
12   #endif
13 } // while(!lcd_clicked)
```

Listing 3.2: Example for Syntax Violating Line-Based Feature Mappings from an Old Version of the Marlin Firmware

To overcome the limitations of the line-based approach, we map features to nodes of an AST as done before in the literature [KAK08]. ASTs are trees describing the syntactic structure of an implementation artefact and are constructed from the grammar of the artefact's language [ALSU06]. In particular, ASTs find usage in compilers for processing source code. As an example, Figure 3.2b shows how an AST represents the syntactical structure of a program fragment, thus revealing the program's hierarchy. Each syntactical element in Figure 3.2a has a corresponding node in the AST in Figure 3.2b. As a fallback, if the implementation artefact does not follow a certain syntax, i.e., no grammar is available, line breaks can be identified to separate elements. Thus, line-based mappings can be expressed as ASTs.

While ASTs ensure syntactic validity, they also require it because they cannot be parsed if the program is not syntactically correct. Hence, only changes leading from a syntactically valid state to another syntactically valid state can be considered. As syntactical correctness of a program is also the goal of developers we do not regard this to be a major disadvantage.

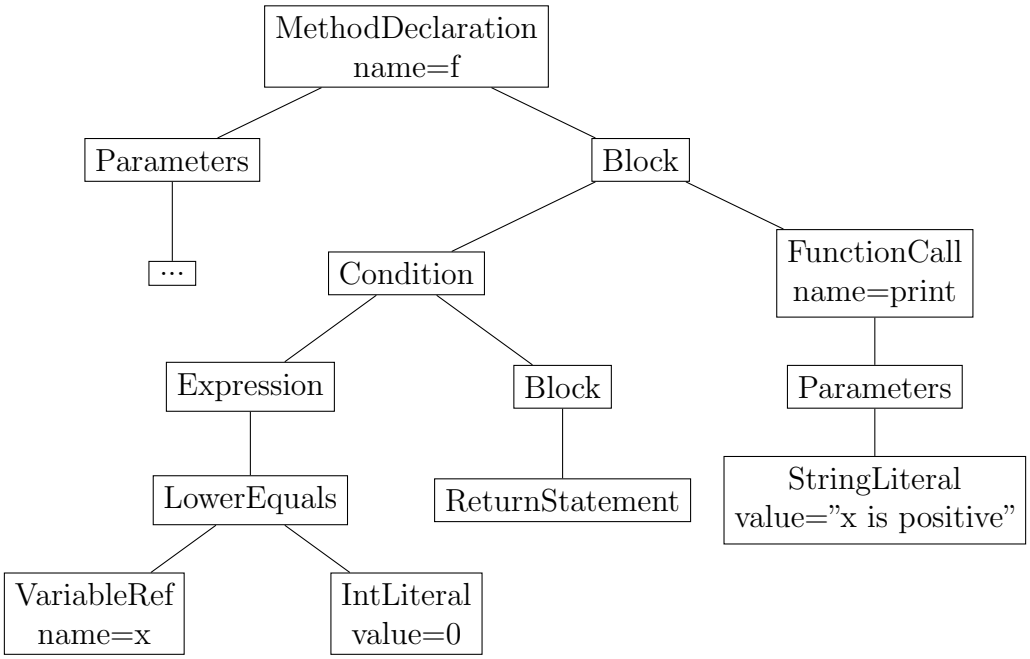
Therefore, we further assume that for any change we consider for our later feature mapping derivation, both versions of the artefact, old and new, are in a syntactically correct state. For now, our derivation of feature mappings from artefact changes is not tied to any specific stage in development, such as the commit stage known from version control systems. Theoretically, our feature mapping derivation can be applied at any time as long as the program can be parsed to an AST.

Using ASTs as feature mapping targets has further advantages in the clone-and-own scenario. As clones are supposed to have differences, independence from code style and order is a convenient property of ASTs. For example, if the order of two function definitions in a source code file is swapped, the AST would remain isomorphic because only the order of nodes representing those function definitions would change. Furthermore, developers do not have to bother annotating syntactical elements, such as commas, because these are not included in ASTs. Consider Listing 3.3, showing function `foo` with two differently mapped parameters. The comma separating both

https://doi.org/10.24355/dbbs.084-202002271120-0

```
void f(int x) {
  if (x <= 0) {
    return;
  }
  print("x is positive");
}
```

(a) Example Java / C++ / C# code



(b) AST of (a)

Figure 3.2: Example of an Abstract Syntax Tree: Each element in the source code (a) has a corresponding node in the tree, constructed according to the language's grammar rules. The parameter node's children are omitted.

https://doi.org/10.24355/dbbs.084-202002271120-0

```
void foo(
  #ifdef A
    int parameterOfFeatureA
    #ifdef B
      ,
    #endif
  #endif
  #ifdef B
    int parameterOfFeatureB
  #endif
);
```

Listing 3.3: Line-Based Mapping of a Feature Interaction Involving the Necessity to Annotate the Parameter Separating Comma

features has to be annotated with both features to prevent syntax errors. With AST-based annotations, it is sufficient to only annotate the parameters. Additionally, the function definitions do not have to be split artificially into multiple lines to allow the mapping definition.

To ensure syntactic validity of variants, only AST nodes should be assignable to features whose deletion does not invalidate the AST.[1] We refer to such nodes as *syntactically optional* nodes. For instance, entire functions or enclosing scopes such as conditions are optional. In contrast, method declarations always require a return type node as their child, and conditions always consist of an expression and a statement block. Hence, these specific child nodes are not *syntactically optional* as they are always required for their parent's definition. Therefore, we refer to them as *syntactically mandatory* nodes. As in CIDE [KAK08], *syntactically mandatory* nodes cannot be mapped to features on their own because removing them invalidates the AST.

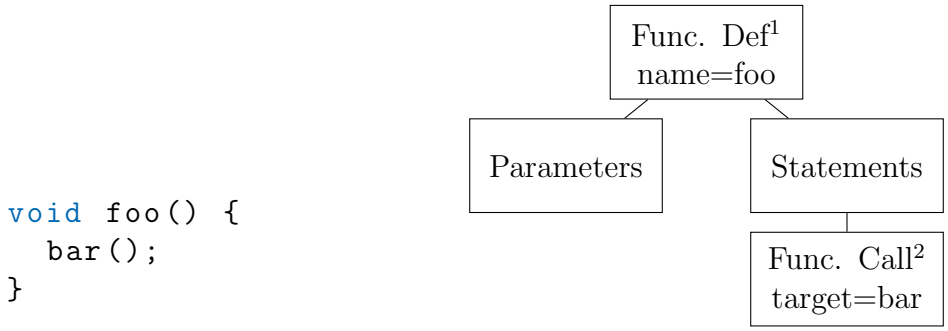
Definition 3.1 (Syntactical Fixture)**.** *An AST node is considered syntactically mandatory, if it is a mandatory part of its parent's definition. Removing it with or without its children leads to a syntactically invalid AST. Otherwise, the node is considered syntactically optional. In essence: Removing a syntactically mandatory node invalidates its parent's definition.*

We refer to nodes being assignable to features as *feature mapping fit*. In that way, *syntactically mandatory* nodes are not feature mapping fit because they cannot be assigned to features. In contrast, *syntactically optional* nodes are feature mapping fit because they can be assigned to features. For now, we only consider the hierarchical structure of an AST for syntactical fixture and no further criteria, such as define-before-use for functions and variables.

As ASTs unveil the hierarchical structure of a program, they allow reasoning on membership of syntactical elements. Artefacts being syntactically dependent on their enclosing structure, such as class or method definitions, can only be present

[1] Deletion in terms of just the single node or even the entire subtree rooted in that node.

https://doi.org/10.24355/dbbs.084-202002271120-0



(a) Implementation of Function `foo` (b) AST of Function `foo`

Figure 3.3: Example Function With Its Corresponding AST

in a variant if the enclosing structure is present. To express this dependency, we propagate feature mappings on AST nodes to all their child nodes as children can only be present when their parent node is present. We refer to this concept as *propagation* of feature mappings in the AST.

Example 3.2.1 (Tree Propagation)**.** The function call to `bar` in Figure 3.3 cannot exist in the program without the enclosing function definition scope of `foo` as a syntax error would be imminent otherwise. Therefore, the function definition has to be present in each variant with the call to `bar`. As shown in Figure 3.3b, this membership is directly reflected in the program's AST. The function call to `bar`, Node 2, is a descendant of Node 1, the definition of function `foo`. Thus, Node 2 can inherit the mapping of Node 1 by concatenating it to its own mapping. For example, if $A, B \in \mathcal{B}$ are the mappings the developer has given to Nodes 1 and 2 respectively, then Node 2 would actually have to be mapped to $A \wedge B$ to not violate the program's syntax. A possible simplification could be made using the global feature model: If $B \Rightarrow A$, then just B as the mapping for Node 2 would be sufficient because the feature model already ensures the necessary inheritance.

Exceptions to the feature mapping propagation are given by embracing scopes whose included statements are not necessarily dependent on them. In particular, not all nodes in the AST are necessarily dependent on all ancestor nodes, i.e., parents, parents of parents, and so on. We refer to nodes that are not mandatory for their children as *hierarchically optional* as deleting and replacing them with their children does not invalidate the AST. Hence, we do not propagate their feature mapping to their children. Contrary, we refer to all other node types as *hierarchically mandatory*.

Example 3.2.2 (Tree Propagation Exception)**.** Consider the code snippet shown in Listing 3.4. It shows code from a virtual reality capable rendering framework being able to run in desktop or virtual reality mode [TSG+19]. Virtual reality is only available if the physical devices are present and software libraries installed, indicated by the guard `WITH_OPENVR`. The software should still be able to run in desktop mode (e.g., for debugging purposes) if the requirements for virtual reality are met. Hence, the desktop setup in Line 9 is independent of the virtual reality feature and thus should not inherit the feature mapping `WITH_OPENVR` of its enclosing `else` branch.

https://doi.org/10.24355/dbbs.084-202002271120-0

```
1   bool isVR = false;
2
3 #ifdef WITH_OPENVR
4   isVR = settings.getBoolOrDefault("isVR", false);
5   if (isVR) {
6     /* virtual reality setup */
7   } else {
8 #endif
9     /* desktop setup */
10 #ifdef WITH_OPENVR
11  }
12 #endif
13 );
```

Listing 3.4: Initialisation Code for the Virtual-Reality Feature in Rendering Framework From [TSG+19]

As for conditions (*if*), we consider other embracing statements as *hierarchically optional*, such as manual scopes (`{ }`) and loops (*while*, *for*, *repeat*). However, some enclosing scopes have to be treated with care. Some of them, such as *for* loops, resource handling scopes (*using*, *with*), and even conditions (*if*) can declare variables used by the enclosed statements, rendering them mandatory indeed. Furthermore, depending on the target language, exception handling (*try*) may or may not be optional. Fortunately, ASTs allow detection of variable declarations in expressions of scopes by searching it in the corresponding subtree.

Definition 3.2 (Hierarchical Fixture)**.** *An AST node is considered hierarchically mandatory, if removing it, such that its children take its place, leads to a syntactically invalid AST. Otherwise, the node is considered hierarchically optional. Removing the entire subtree can still be possible in both cases. In essence: Removing a hierarchically mandatory node orphans all its descendants (children, grandchildren, etc.) in the textual representation.*

To clarify the relationship between the properties *hierarchically mandatory*, *syntactically mandatory*, and their optional counterpart, Table 3.1 summarises their possible combinations. AST nodes only propagate their feature mapping, if they can have a feature mapping, i.e., are *syntactically optional*. For instance, removing an entire class or function does not invalidate an AST. However, *syntactically optional* nodes that are *hierarchically optional* do not propagate their feature mapping, as they can be removed from the tree while their children are kept. The only nodes in this category are the enclosing scopes, such as conditions and loops, as explained before.

However, in the clone-and-own scenario, mapping the mandatory return type of functions can be useful, as return types can differ across software clones. Though, such mappings of mandatory nodes have to be treated with care, as they need a valid mapping for each possible configuration, such that an instance of the mandatory node is available for each clone. When a new clone is introduced, a return type could

https://doi.org/10.24355/dbbs.084-202002271120-0

AST node is		hierarchically			
		optional		mandatory	
syntactically optional		Fitness	yes	Fitness	yes
		Propagation	no	Propagation	yes
		Examples	non-defining enclosing scopes	Examples	function definition, class definition
syntactically mandatory		Fitness	no	Fitness	no
		Propagation	no	Propagation	no
		Examples	block nodes below enclosing scopes, return types	Examples	block node of function

Table 3.1: Overview on Feature Mapping Fitness and Propagation of Abstract Syntax Tree Node Types

be recommended from existing clones according to the new clones configuration. We do not consider such exception in detail furthermore but as they are orthogonal to our feature mapping requirements we could still integrate such cases later.

3.2.2 Granularity of Annotations on Abstract Syntax Trees

Upon annotating ASTs, an important question is the granularity of feature mappings. Different software projects may require different levels of granularity. For instance, frequently used preprocessor annotations even allow annotating parts of names. Annotating source code with preprocessor statements in a maintainable and readable fashion is denominated as *disciplined annotations* [KAT+09, LKA11]. Annotations are considered disciplined if deletion of annotated artefacts does not invalidate a program's syntax. Thus, disciplined annotations are a subset of all possible annotations. As our classification in *syntactically optional* and *syntactically mandatory* AST nodes ensures syntactic validity upon removing annotated nodes, our feature mappings are also disciplined annotations.

By annotating the rules of a language's grammar with variability information, ASTs can be constructed in any desired granularity and thus allow feature mappings in any granularity. Apel et al. [AKL13] introduced this approach and used it on stripped down ASTs tracing a single feature throughout a whole software project, so called Feature Structure Trees (FSTs). Thus, granularity of feature mappings can be seen as a parameter adjustable to any project's needs. Detecting a suitable level of granularity for an individual software project however requires careful analysis and annotation of the grammar which is a complicated, error-prone, and time-consuming task. Furthermore, disciplined annotations already restrict the granularity to more-coarse grained structures. For this thesis, we consider grammars of common programming languages (e.g., Java, C#, C++, Python, etc.) as a basis for AST construction.

https://doi.org/10.24355/dbbs.084-202002271120-0

Similar to FSTs, we also need to represent a whole software project in a single tree to be able to detect moves of artefacts. For instance, during refactoring it is common for developers to move certain functions or classes to different locations, e.g., files or packages. Thus, we create a single large *project structure tree* reflecting the structure of artefacts in an entire variant. In clone-and-own development, each variant corresponds to exactly one project structure tree. As shown in Figure 3.4, the project structure tree consists of *directory* and *file* nodes[2], called *project structure nodes*. Each file node has the ASTs of the implementation artefact it contains as its child. As removing an entire directory or file from the project does not invalidate the syntax of the program, project structure nodes are *syntactically optional*. However, removing such a file node while keeping its child AST, invalidates the tree because implementations are bound to files containing them. Therefore, file nodes are *hierarchically mandatory*. Although removing directory nodes while keeping the child file nodes does not invalidate the project structure tree, we also consider it as *hierarchically mandatory* for consistency and the opportunity to assign whole software modules to a single feature. Consequently, according to Table 3.1 project structure nodes are feature mapping fit, i.e., can be assigned to features, and propagate these mappings to their children. As project structure nodes do not require exceptions or extensions to our AST-based feature mappings, we consider them to be part of our ASTs. From now on, when referring to ASTs this implicitly includes the whole project structure tree.

3.3 Differencing of Abstract Syntax Trees

To derive feature mappings upon software changes, we first have to identify how software artefacts can be changed and how these changes are represented in the corresponding AST. This enables us to incorporate the types of edits developers make during programming. Our goal thereby is, to reflect the developer's intent on changes more accurately when deriving feature-mappings in a satisfying semi-automated fashion.

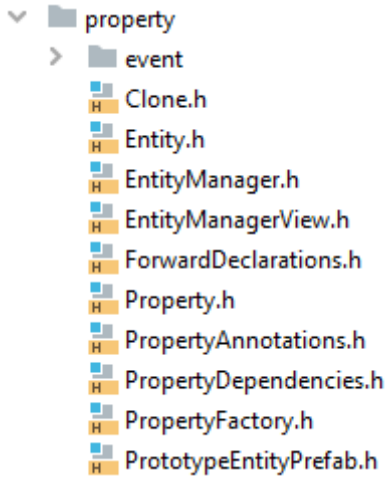
We distinguish four types of changes: *insertion*, *deletion*, *update*, and *move*. To avoid disambiguities, we refer to these types as *edit operations* and to concrete applications of them as *edits*. Although any changes on artefacts can be expressed with insertions and deletions only, being able to classify changes as updates or moves allows substantially more accurate reasoning on edits. For instance, if we would only consider insertions and deletions, renaming a class would require to delete and reinsert an entire subtree in the AST, as the children of the class node cannot exist without it as it is *hierarchically mandatory*. Such coarse-grained change definitions are inadequate for reasonable change analysis.

To formally define edit operations on ASTs, we first give a formal definition of ASTs on which we operate.

Definition 3.3 (Set of Artefacts)**.** *Let $\mathcal{A}$ be the universe of all software artefacts. It is a set containing all possible source code lines, AST nodes and other artefacts belonging to software project.*

[2]Depending on the target language and project setup these could also be considered to be *module* or *package* nodes.

https://doi.org/10.24355/dbbs.084-202002271120-0



(a) Project Layout in File System

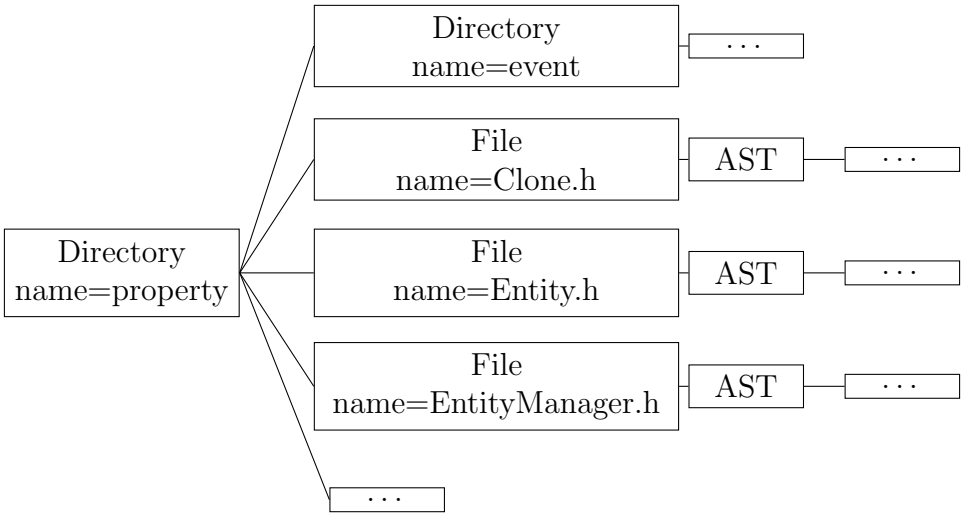


Figure 3.4: Example for Project Structure Tree Derived From File System

Definition 3.4 (Set of Abstract Syntax Tree Nodes)**.** *Let $\Uparrow \subset \mathcal{A}$ be the universe of all AST nodes. It is a set containing all possible AST nodes of any AST. Therefore, the nodes of a concrete AST are always a subset of $\Uparrow$.*

We use these two definitions to classify software artefacts as such.

Definition 3.5 (Abstract Syntax Tree)**.** *An AST is a labeled ordered directed acyclic graph $T = (V, E, r, I, \Sigma, \mathrm{L})$ with:*

- *nodes $V \subset \Uparrow$. We abbreviate $v \in T$ for any node $v \in V$. A node can potentially be part of multiple ASTs, for instance the old and new version of an AST before and after a change by the developer.*
- *root node $r \in V$. We write $root(T) := r$.*
- *parent-child relationship $E \in V \times V$. A node $p \in V$ is considered to be the parent of node $v \in V$, if and only if $(p, v) \in E$, denoted by $parent_T(v) = p$. Each node has exactly one parent except for the root:*

$$\forall v \in V.\ v \neq r \Leftrightarrow \exists! p \in V.\ p = parent_T(v)$$

https://doi.org/10.24355/dbbs.084-202002271120-0

- *index function* $I : V \to \mathbb{N}$ *ordering the children of a node. Each node gets an index that indicates its location under its parent node from left to right. Thereby, indices between children of the same node are unique because they define a total order on those children. The root node is the only node without an index.*

- *set* Σ *of node types according to the underlying grammar rules, such as* `MethodDeclaration`, `Class`, *or* `Condition`.

- *label function* $\mathrm{L} : V \to \Sigma \times \mathrm{String}$ *assigning each node a type and a value. For instance, the AST node* v *representing the function declaration* `foo` *would have type and label* $\mathrm{L}(v) = (\texttt{FunctionDeclaration}, \texttt{foo})$.

Let $child_T(v, i)$ *be the* i*-th child of node* v *in the tree* T*, i.e., the node* c *for which* $v = parent_T(c)$ *and* $I(c) = i$*. The transitive closure of* E *on a node* $v \in V$ *delivers all its descendants, denoted by* $child^*_T(v)$*. Thus, for any tree* $T = (V, E, r, I, \Sigma, \mathrm{L})$*, let* $tree(v) = (\{v\} \cup child^*_T(v), E, v, I, \Sigma, \mathrm{L})$ *denote the subtree rooted in* $v \in V$*. For any subtree* S *of tree* T *we write* $S \subseteq T$*. From our definitions follows that* $tree(v) \subseteq T$ *for each* $v \in T$ *and* $T = tree(root(T))$.

Definition 3.6 (Set of all Abstract Syntax Trees)**.** *Let* $\Uparrow^*$ *denote the set of all ASTs, i.e., any AST* T *is an element of this set* $T \subset \Uparrow^*$.

Definition 3.7 (Edits on Abstract Syntax Trees)**.** *An edit* e *is an instance of an edit operation, transforming an AST* T_1 *to another AST* T_2*. We write* $T_1 \xrightarrow{e} T_2$.

Such an edit can be arbitrarily complex and may itself consist of several smaller edits as we show in the following sections.

To ensure syntactical correctness of a program upon feature removal we follow the paradigm of disciplined annotations as explained in Section 3.2.2. Therefore, we allow feature mappings for *syntactically optional* AST nodes only. For instance, such nodes are structures, such as enclosing scopes (*if*, *while*, *for*), or function or class definitions. We refer to such nodes as feature mapping fit. However, some feature mapping fit nodes may still be mandatory for their children. For instance, the statements within a function definition cannot exist without the function definition scope. Thus, children of such *hierarchically mandatory* nodes should receive their parents mapping to express this dependency. An overview to these classification is given in Table 3.1 on Page 22. We define propositional predicates for identification of feature mapping fit and propagating node types:

Definition 3.8 (Feature Mapping Fitness)**.** *For an AST with alphabet* Σ*, let the propositional predicate* $\mathrm{Fit} : \Sigma \to \{\mathrm{false}, \mathrm{true}\} \subset \mathcal{B}(\emptyset)$ *denote the fitness of node types for feature mappings, i.e., if they are assignable to features as shown in Table 3.1 on Page 22.*

Definition 3.9 (Feature Mapping Propagation)**.** *For an AST with alphabet* Σ*, let the propositional predicate* $\mathrm{Propagates} : \Sigma \to \{\mathrm{false}, \mathrm{true}\} \subset \mathcal{B}(\emptyset)$ *denote if a node type propagates its feature mappings to its children as shown in Table 3.1 on Page 22.*

https://doi.org/10.24355/dbbs.084-202002271120-0

As a shorthand, we also write Propagates(a) for Propagates(t) where $t \in \Sigma$ is the type of node $a \in \pitchfork$. Nodes can only propagate their feature mapping, if they have one. Therefore, propagating nodes must be feature mapping fit:

$$\forall \sigma \in \Sigma\,.\ \text{Propagates}(\sigma) \models \text{Fit}(\sigma). \tag{3.1}$$

For further use, let $\widehat{parent}_T(a) \coloneqq p$ denote the nearest ancestor $p \in T$ of a node $a \in T$ in the tree T with $\models$ Propagates(p). If no such ancestor exists, $\widehat{parent}_T(a) \coloneqq \varepsilon$, where $\varepsilon \in \mathcal{A}$ denotes an undefined value, such as `NULL` does for many programming languages.

In the following, we first describe how edits on source code are represented on the AST layer and thereby define the different edit operations formally. Second, we elaborate on how edits can be identified on AST differences (diffs), i.e., which sequence of edits transforms a given AST into another one. Third, we argue how semantic lifting can be used to detect user-level edits in a set of fine-grained low-level edits on ASTs.

3.3.1 Semantic Edits on Abstract Syntax Trees

Editing software artefacts in their text-based representation leads to corresponding transformations on their ASTs. While developers pursue a certain intent with their changes, neither text-based nor AST-based representations of the edits need to reflect these intents in an intuitive or understandable way. Thus, if developers would classify a change they made as an insertion, we want to recognise it as such on the ASTs. We refer to such edits as *semantic edits* as they do not only encode syntactical changes but reflect the developer's intent. In the following, we derive our definitions of semantic edit operations on ASTs from a reasonable intuition on edits in the text-based representation:

Insert Some node types cannot exist without certain children they require for their definition. For example, conditions always require an expression and a block node as their children and binary expressions always require exactly two subexpressions[3] as shown in Figure 3.5d. Inserting the `Condition` node together with its children `Expression` and `Block` has to be considered as an atomic operation, as it would result in ill-formed intermediate ASTs otherwise. For that reason, we allow the insertion of whole subtrees at once:

$$\mathit{insert}_{\text{tree}} : \pitchfork^* \times \pitchfork^* \times \pitchfork \times \mathbb{N} \to \pitchfork^*, \tag{3.2}$$

where $\mathit{insert}_{\text{tree}}(T, U, p, i) = T'$ adds the tree $U \not\subseteq T$ as the i-th child of $p \in T$, such that $U \subset T'$. Existing children of p with index greater than i are shifted to the right, i.e., their index is increased by 1. Inserting single leaves to the tree can also be expressed with $\mathit{insert}_{\text{tree}}$ because a single node is also a tree. Nevertheless, for later reference, we introduce $\mathit{insert}_{\text{node}}$ to explicitly refer to $\mathit{insert}_{\text{tree}}$ with just a single node.

[3] Such constraints directly emerge from a language's grammar and thus are language dependent. However, most conceptual constraints are similar between programming languages.

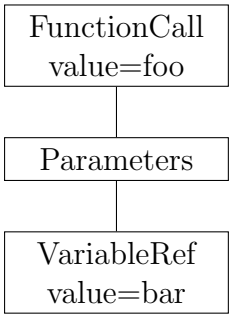
https://doi.org/10.24355/dbbs.084-202002271120-0

```
foo(bar);
```

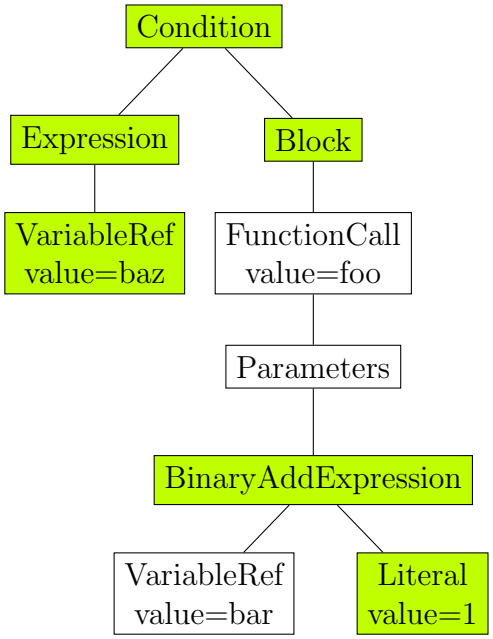
(a) Call `foo`

```
if (baz) {
  foo(bar + 1);
}
```

(b) Call `foo` with Insertion of a Condition and an Increment in the Argument



(c) AST of (a)



(d) AST of (b): Nodes added to the original AST (c) are marked green.

Figure 3.5: Insertions in Implementation Artefacts Need Not to Correspond to Sole Insertions (Of Leaf Nodes) in the AST: An enclosing condition as well as an increment to the parameter are inserted into an existing code fragment (a) leading to a new fragment (c). Although only code insertions were performed, the AST (b) got re-rooted and split in the middle.

https://doi.org/10.24355/dbbs.084-202002271120-0

Furthermore, insertions in implementation artefacts need not to correspond to subtree insertions in ASTs only. As an example, Figure 3.5 shows how pure insertions in implementation artefacts result in structural changes beyond tree insertion in the AST. The code Figure 3.5a is changed to Figure 3.5b, and thereby transforms the corresponding AST from Figure 3.5c to Figure 3.5d. Accordingly, insertions in implementation artefacts allow new nodes to replace existing nodes. In our example, the `BinaryAddExpression` replaces the original variable reference to `bar`. The `Condition` node replaces the `FunctionCall` node and relocates it even further down the hierarchy. Moreover, there could be more children next to `FunctionCall`, e.g., other statements that have to be moved to the `Block` node's children. To detect artefact insertion more accurately, we consider the above described replacement and adoption of existing nodes:

$$insert_{\text{partial}} : \pitchfork^* \times \pitchfork^* \times \pitchfork \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \pitchfork \to \pitchfork^*, \tag{3.3}$$

where $insert_{\text{partial}}(T, U, p, i, j, k, u)$ with $i \leq j$ replaces the children of node $p \in T$ in range i to j with the new subtree $U \not\subseteq T$, then located at index i. The replaced children are added as children of $u \in U$ at index k.

For our later feature mapping derivation, we consider both $insert_{\text{tree}}$ and $insert_{\text{partial}}$ as insertions.

Delete Deletions on implementation artefacts are not restricted to AST leaf nodes, as for instance the deletion of an enclosing scope. Furthermore, depending on the granularity of ASTs, different implementation artefacts are represented as leaf nodes. Therefore, we allow deleting any *syntactically optional* non-root node instead of just leaves:

$$delete_{\text{node}} : \pitchfork^* \times \pitchfork \to \pitchfork^*, \tag{3.4}$$

where $delete_{\text{node}}(T, v)$ deletes node $v \in T$ with $v \neq root(T)$ and $\models \text{Fit}(v)$. The children of v are inserted as children of $parent_T(v)$ in the place v was before. However, not all nodes can be deleted on their own as they may be *syntactically mandatory*, such as block and expression of conditions. Hence, removing a set of nodes in syntax preserving fashion is needed.

Removing entire methods, classes, or scopes can be expressed through consecutive $delete_{\text{node}}$ edits. However, these have to be ordered to avoid unnecessary or even impossible child rearranging. Therefore, we define a second delete operation, deleting a whole subtree with *syntactically optional* root:

$$delete_{\text{tree}} : \pitchfork^* \times \pitchfork \to \pitchfork^*, \tag{3.5}$$

where $delete_{\text{tree}}(T, r)$ deletes the whole subtree rooted in $r \in T$ with $r \neq root(T)$ and $\models \text{Fit}(r)$, i.e., r is syntactically optional.

Nevertheless, there are still cases of deletions in the source code that can neither be expressed with a single $delete_{\text{node}}$ nor $delete_{\text{tree}}$. Consider the inversion of the insertions shown in Figure 3.5. Removing the condition from 3.5b results in the deletion of the partial subtree rooted in the `Condition` node. As this is the inverse operation of the above described $insert_{\text{partial}}$ we define

$$delete_{\text{partial}} : \pitchfork^* \times \pitchfork \times \mathcal{P}(\pitchfork) \to \pitchfork^*, \tag{3.6}$$

https://doi.org/10.24355/dbbs.084-202002271120-0

where $delete_{\text{partial}}(T, r, \{v_1, \ldots, v_k\})$ deletes the subtree rooted in $r \in T$ but without the subtrees rooted in descendants $v_i \in child_T^*(r),\ i \in \{1, \ldots, k\}$. The descendants are delegated to $parent_T(r)$ in the place r was before. Note that r can also be the root of T. In this case exactly one descendant has to be saved from deletion, i.e., $k = 1$, as there has to be a root and there cannot be more than one root. In contrast to $delete_{\text{node}}$ and $delete_{\text{partial}}$, the root r does not need to be *syntactically optional* as saving the children $\{v_1, \ldots, v_k\}$ can lead to a valid AST again. As an example, consider Figure 3.6, where the `BinaryAnd` is moved. Although the `BinaryAnd` is not *syntactically optional* here as it is required by the above expression, it can be deleted in terms of $delete_{\text{partial}}$ because `Literal` with name b fills in the emerging gap.

For our later feature mapping derivation, we consider $delete_{\text{node}}$, $delete_{\text{tree}}$, and $delete_{\text{partial}}$ as deletions.

Update Some changes on implementation artefacts do not change the AST's structure. For example, renaming a function or changing its return type does neither affect the function's structure nor the elements it contains. We refer to such edits as updates. Both, type and value of a node can be changed:

$$update : \Lambda^* \times \Lambda \times (\Sigma \times \text{String}) \to \Lambda^*, \tag{3.7}$$

where $update((V, E, r, I, \Sigma, \text{L}), v, (t, s)) = (V, E, r, I, \Sigma, \text{L}')$ updates node $v \in V$ to have type t and value s:

$$\text{L}'(w) = \begin{cases} (t, s), & v = w, \\ L(w), & \text{else.} \end{cases}$$

Updates can be expressed through inserting and deleting the corresponding subtree. Though, our goal is to reflect the developers intentions as accurately as possible. Considering updates as self-contained operations avoids obfuscated diffs and instead leads to more clear change descriptions. Furthermore, it allows us to incorporate existing feature mappings of updated nodes that would get lost when representing the update with a deletion and insertion. Detecting renamings as such could enable more reasonable variant synchronisation as the renaming could be consistently performed in target variants instead of synchronising single occurrences of changes each.

Move Each implementation artefact has a corresponding subtree in the AST of the program it is contained in. For example, a method definition is represented by the AST subtree rooted in its corresponding `MethodDefinition` node. Therefore, whenever an implementation artefact is moved in a syntax preserving fashion, its subtree is moved in the AST. Moving a subtree thereby means to remove it from the tree and insert it again as the child of another node. We define

$$move_{\text{tree}} : \Lambda^* \times \Lambda \times \Lambda \times \mathbb{N} \to \Lambda^*, \tag{3.8}$$

where $move_{\text{tree}}(T, v, p, i)$ removes the subtree rooted in $v \in T$ and inserts it as the i-th child of $p \in T$.

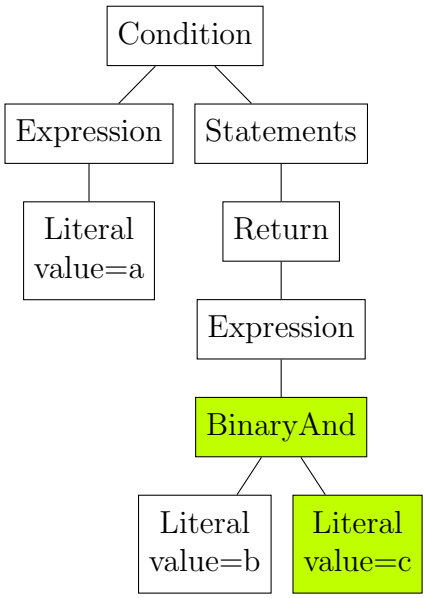
https://doi.org/10.24355/dbbs.084-202002271120-0

```
if (a) {
  return b && c;
}
```

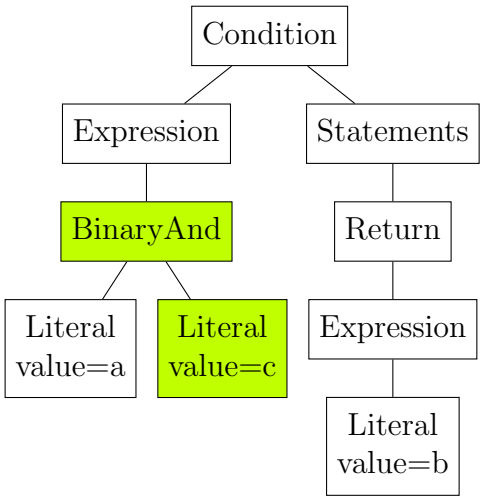
(a) Initial Condition

```
if (a && c) {
  return b;
}
```

(b) Condition With Moved Binary-And Expression



(c) AST of Condition (a)



(d) AST of Condition (b)

Figure 3.6: Move of Partial Subtree in AST

https://doi.org/10.24355/dbbs.084-202002271120-0

However, in more advanced cases, subtrees can be split such that they are only partially removed while leaving children behind. Consider, Figure 3.6. Moving the code fragment "`&& c`" is represented as removing a partial subtree like $delete_{\text{partial}}$ does and reinserting it in the way $insert_{\text{partial}}$ does. Hence, let

$$move_{\text{partial}} : \pitchfork^* \times \pitchfork \times \pitchfork \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathcal{P}(\pitchfork) \times \pitchfork \to \pitchfork^*, \tag{3.9}$$

where $move_{\text{partial}}(T, v, p, i, j, k, L, u) = insert_{\text{partial}}(T', V', p, i, j, k, u)$ describes the move of the partial subtree V' rooted in $v \in T$ whereby V' is exactly the subtree deleted by $delete_{\text{partial}}(T, v, L) = T'$.

Moves can be expressed through inserting and deleting the corresponding (partial) subtree, i.e., through $delete_{\text{partial}}$ and $insert_{\text{partial}}$. Considering moves as self-contained operations however, allows the conservation of existing mappings which would get lost during deletions and reinsertions. Thus, when identifying text operations as reasonable moves in the AST, the original feature mappings of moved nodes can be kept or updated.

For our later feature mapping derivation, we consider both $move_{\text{tree}}$ and $move_{\text{partial}}$ as moves.

To reflect developers' intentions on code changes in even more detail, more semantic edit operations could be considered. The *update* edit operation for example only works for single nodes so far and does not reflect coherent changes in multiple locations, such as refactoring the class name across the entire code base. Detecting renaming of structures could ease variant synchronisation, as such a renaming operation could be executed on target variants instead of synchronising a set of local updates. This would circumnavigate the problem that not all updates may have a target in other variants and that some locations in the target variants would not get updated because they do not have a counterpart in the source variant.

3.3.2 Deriving Abstract Syntax Tree Edit Scripts

During software development, implementation artefacts evolve, originating in a sequence of versions. The differences between two versions can be expressed as a series of changes which transform an artefact to its new version. In the literature, such sequences of edits are referred to as *edit scripts* [Bil05, FMB+14, CRGMW96]. To detect the semantic edits applied to artefacts, we need to recover edit scripts on ASTs.

Definition 3.10. *An edit script $\mathscr{E}$ transforming an AST T to another AST T' is a sequence of edits $(e_1, e_2, \ldots, e_n)$, such that $T \xrightarrow{e_1} T_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} T'$. As an edit script itself is an edit, we also write $T \xrightarrow{\mathscr{E}} T'$.*

Recovering edit scripts is a well researched topic [Bil05, PA11, FMB+14, CRGMW96, HM08]. Commonly, two steps are involved in edit script computation:

1. Match Detection: To detect similarities between both trees, a matching is computed. A match consists of exactly two nodes of different trees, where both nodes are considered to be equal. Each node is part of at most one match. A matching the set of all such matches for two trees.

https://doi.org/10.24355/dbbs.084-202002271120-0

2. Differencing: An edit script is deduced using the matching from the previous step. For unmatched nodes, no similar node in the other tree could be found. These nodes are considered to emerge from edits. Unmatched nodes in the source tree are usually considered to be deleted as they do not occur in the target tree. Correspondingly, unmatched nodes in the target tree are usually considered to be inserted. Advanced heuristics are used for detecting moves and updates on unmatched nodes. The computed edit script depends on the matching quality.

To compute optimal edit scripts, the different edit operations are assigned a cost. For example, deletion and insertion could be considered more expensive than an update because changing a node's label may be more likely than removing it and inserting an altered version of it.

Throughout the literature, different definitions for the edit operations exist. We compare existing definitions of the algorithms *LaDiff* by Chawathe et al. [CRGMW96], *Diff/TS* by Hashimoto and Mori [HM08], *RTED* by Pawlik and Augsten [PA11], *GumTree* by Falleri et al. [FMB+14] and the definitions by Bille, he used in his survey [Bil05] in Table 3.2 on Page 33. Across the five considered works, different definitions for the four basic operations are present. Insertions in terms of $insert_{\text{tree}}$ or $insert_{\text{partial}}$ are used across all methods but on single nodes instead of whole subtrees. As GumTree enhances the matching of LaDiff but uses it for edit script computation both have the exact same definitions. They define *delete* as the deletion of a leaf node, whereas Bille's definitions and Diff/TS allow deletion of any non-root node, delegating its children to its former parent, as our $delete_{\text{node}}$ does. As for insertions, none of them allows deleting an entire subtree at once. The move operation is only considered in three works and is the same as ours for two of them. The rather uncommon definition of the move operation by Diff/TS is the same as our $move_{\text{partial}}$ (and inspired our definition of $move_{\text{partial}}$). Whereas we assign type and value to each AST node, these are usually referred to as *label* and *value* in the literature. Bille's definitions, Diff/TS, and RTED only assign a single label to each node, which is equivalent to having label and value, in theory. LaDiff and GumTree assign label and value to each node but allow updating values only. They consider, the node type (i.e., its label) to be constant.

More work focuses on improving the GumTree algorithm. Matsumoto et al. improve the matching phase with pre-processed line-based diffs to identify unchanged lines in advance [MHK19]. Dotzler and Philippsen introduce five different optimisations usable for various tree differencing algorithms but directly compare it to GumTree only [DP16].

To reuse existing tree differencing algorithms, we need to detect more sophisticated user-level edits in the mostly technically motivated edit scripts they compute. In the following section, we show how *semantic lifting* could be used to recover edits matching our definitions from low-level edit scripts, computed by existing algorithms. This enables us to classify types edits more accurately to better reflect developers' intentions.

https://doi.org/10.24355/dbbs.084-202002271120-0

Work	Insert	Delete	Move	Update
LaDiff by Chawathe et al. [CRGMW96]	$insert_{\text{node}}$	$delete_{\text{node}}$ on leaves	$move_{\text{tree}}$	value only
Bille's definitions [Bil05]	$insert_{\text{partial}}$ a single node	$delete_{\text{node}}$	-	nodes have label only
Diff/TS by Hashimoto and Mori [HM08]	$insert_{\text{partial}}$ a single node	$delete_{\text{node}}$	$move_{\text{partial}}$	nodes have label only
RTED by Pawlik and Augsten [PA11]	$insert_{\text{node}}$	$delete_{\text{node}}$	-	nodes have label only
GumTree by Falleri et al. [FMB+14]	$insert_{\text{node}}$	$delete_{\text{node}}$ on leaves	$move_{\text{tree}}$	value only
Definitions we desire	($insert_{\text{node}}$,) $insert_{\text{tree}}$, $insert_{\text{partial}}$	$delete_{\text{node}}$, $delete_{\text{tree}}$, $delete_{\text{partial}}$	$move_{\text{tree}}$, $move_{\text{partial}}$	*update*

Table 3.2: Comparison of Definitions of Common Tree Operations Throughout the Literature: A dash (-) indicates that an operation is not considered in the corresponding work. If the literature's definitions match ours or are weaker, we describe them with our definitions.

3.3.3 Semantic Lifting of Abstract Syntax Tree Edit Scripts

The edit operations used in the literature are very fine-grained and designed according to the internal software representation as ASTs. However, developers work with and are accustomed to external representations, such as text or models. Thereby, the conceptual edits they make follow a certain intent and thus are usually more coarse-grained as discussed in the previous sections. Hence, edit scripts recovered by existing tree or model diffing tools rarely present changes in an understandable or intuitve way. Instead, semantic edits are decomposed to numerous low-level changes.

Detecting coarse-grained conceptual edits in low-level edit scripts was first addressed by Kehrer et al. and referred to as *semantic lifting* [KKT11]. Figure 3.7 shows how state-of-the-art edit script detection algorithms are extended by the semantic lifting post-processing. After an edit script of low-level tree operations is computed, semantic lifting identifies user-level edit operations developers are able to use in that edit script.

Semantically lifting an edit script means to partition an edit script into disjoint subsets of low-level edits implementing a larger semantic edit operation. Such subsets are referred to as *semantic change sets* [KKT11]. Semantic change sets have to be disjoint as each low-level tree edit results from exactly one user-level operation on the source code.

https://doi.org/10.24355/dbbs.084-202002271120-0

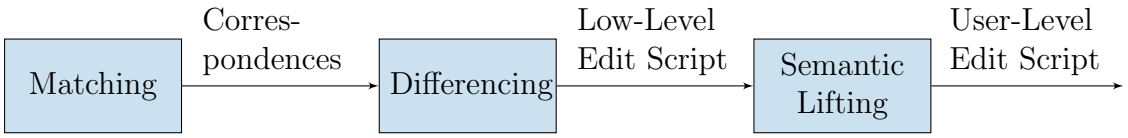


Figure 3.7: Abstract Syntax Tree Differencing Pipeline With Semantic Lifting [KKT11]: By extending the state-of-the-art pipeline for edit script recovery with semantic lifting, low-level changes are grouped to operations users work with.

Definition 3.11 (Semantic Change Sets)**.** *For an edit script $\mathscr{E} = (e_1, \ldots, e_n)$ a sequence $S = (s_1, \ldots, s_m)$, $m \leq n$ with $s_i \subseteq \{e_1, \ldots, e_n\}$, $i \in \{1, \ldots, m\}$ is a sequence of semantic change sets, if and only if $s_i \cap s_j = \emptyset$ for all $1 \leq i < j \leq m$ and $\bigcup_{i \in \{1,\ldots,m\}} s_i = \{e_1, \ldots, e_n\}$. The edits contained in a semantic change set are ordered implicitly by their index.*

Kehrer et al. match pre-defined patterns against edit scripts to identify semantic change sets [KKT11]. First, a pattern that is to be recognised in the edit script is defined. Afterwards, according to the patterns, the low-level changes are grouped to semantic change sets. An operation handling exactly those two steps for a single specified semantic edit, is referred to as a *change set recognition rule*.

Existing work on semantic lifting is focused on model-driven software development [KKT11, KKOS12, bKLW12] or even business process modelling [NMLT08]. In model-driven software development, models (usually represented as UML class diagrams), are primary development artefacts. These models exhibit a graph structure, wherefore they can be interpreted as abstract syntax graphs similar to ASTs for source code. As abstract syntax graphs are a more general concept than ASTs, concepts from semantic lifting on them could be reused. In the model-driven software development context, semantic change set rules are derived automatically from their corresponding graph transformation rule being available to the user in their model representation, e.g., a graphical editor.

To our knowledge, no work exists on semantic lifting of AST edit scripts. Thus, semantic change set rules for ASTs have to be identified yet. Rules from the model-driven software development context cannot be reused because change set rule specification strongly depends on the underlying data, e.g., model or AST. Hence, we derived an initial subset on edit operations that we consider semantically reasonable in the last section. However, those operations cannot be translated automatically to semantic change set rules as for model editing.

Definition 3.12 (Semantic Edit Scripts)**.** *An edit script $\hat{\mathscr{E}}$ is considered semantic edit script, if and only if it at least contains a single semantic edit.*

Further challenges for semantic lifting are the potential arbitrary order of commutative edits as well as incompleteness of edit scripts. For instance, a deletion and an insertion forming a move could be separated from each other by an arbitrary amount of other independent intermediate edits in-between them in the edit script to lift. Thus, semantic lifting has to detect the commutativity of certain edits. The same scenario can be used to explain the problem of incomplete edit scripts. Removing a

https://doi.org/10.24355/dbbs.084-202002271120-0

function from the code base is considered a deletion upon tree diffing and semantic lifting. If however the function is reinserted again after change analysis, neither tree diffing nor semantic lifting are able to detect that operation as a move but only as an insertion because the function's deletion was already handled during the last diffing. Thus, a mechanism for consistent backwards-compatible semantic lifting is necessary. A possible approach could be the retrospective consideration of the entire history of changes. Semantic lifting on ASTs exceeds the scope of this thesis. We elaborate further on it in Chapter 9.

3.4 Summary

In this chapter, we introduced semantic edits on ASTs. First, we identified ASTs as reasonable feature mapping targets to implement *disciplined annotations* [KAT+09]. To conform to the intentions of developers on edits more accurately, we introduced *semantic edits*. We explained why existing work on tree differencing itself is insufficient for semantic change detection. *Semantic lifting* could detect semantic changes in low-level tree diffs produced by existing algorithms.

Our insights are relevant for user-oriented tree differencing and thereby software (product-line) development in general as our assumptions on clone-and-own, introduced in Section 3.1, are reasonable and universal enough. First, we assume clones to originate from each other, i.e., that they have commonalities and deliberately introduced differences. Second, developers agree on a common domain of features they target to implement. Third, it is known which features are implemented in which variant. It is not necessary that this knowledge is given by a single developer. Fourth, during development, developers typically know which feature or feature interaction they are currently working on.

To support feature interactions, we use propositional formulas for feature mappings instead of sole features. As opposed to line-based feature mappings, ASTs allows reasoning on an syntactic structure of artefacts. This bears several advantages but mainly preserving syntactic validity upon feature composition, also known as disciplined annotations [KAT+09]. We ensure syntactic validity by propagating feature mappings of AST nodes to all descendants in the hierarchy depending on certain properties of that node. Therefore, we formally summarised existing AST-based feature mappings by Kästner et al. [KAK08].

For our later feature mapping derivation upon edits of implementation artefacts, we investigated how changes in source code affect the AST. Thereby, we synthesised *semantic edits*, eight edit operations on ASTs that coincide with semantically reasonable edits in the textual representation of software artefacts. We classify our eight semantic edits as either insertion, deletion, move, or update for our following feature mapping derivation.

We have shown that existing work on tree differencing does not consider semantic edits yet but instead computes edit scripts of low-level technical operations on trees. We elaborated how a technique known as semantic lifting, first used in model-driven software development [KKT11], can be partially reused for semantic diffing on trees in general. As developing semantic lifting is out of scope of this thesis, we will work in this topic in the future as stated in Chapter 9.

https://doi.org/10.24355/dbbs.084-202002271120-0

In the next chapter, we show how feature mappings can be derived upon tree edit operations. For each type of edit (i.e., insertion, deletion, move, and update) we individually develop a function, deriving the mappings of an edited AST from its existing mappings and developer's domain knowledge.

https://doi.org/10.24355/dbbs.084-202002271120-0

4. Semi-Automated Feature Mapping Recording Upon Semantic Edits

In this chapter, we introduce our approach for recording feature mappings during software development in a semi-automated fashion. Therefore, we use the abstract representation of implementation artefacts introduced in the last chapter to obtain knowledge on their hierarchical structure. By incorporating the developer's domain knowledge on edited features, we derive the feature mapping of an artefact's new version upon semantic edits. Thereby, we distinguish between the intuitive high-level changes insertion, deletion, move, and update to comply developers' intentions in a reasonable way.

During software development, i.e., when editing any kind of software artefact, developers specify on which feature(s) they are working on as a propositional formula called the *feature context*. All artefact changes should belong to the currently active feature context. We investigate how to deal with changes to already mapped artefacts under a given feature context. Depending on the type of change, e.g., insertions or deletions, developers could express different intentions with their feature context and even children in the implementation hierarchy could be influenced.

In Section 4.1, we formally introduce feature context, feature mappings, and our algorithm for feature mapping derivation from semantic edit scripts on ASTs. After illustrating general constraints on feature mappings, we deduce our semi-automatic derivation of feature mappings during software development, especially taking care of changes to already mapped artefacts. In Section 4.2, we show how to involve the global feature model for derivation and simplification of feature mappings. In Section 4.3, we give an outlook how other variants can be incorporated for obtaining knowledge on feature mapping derivation. We present two artificial exploits in Section 4.4 that allow mapping features arbitrarily from a single source variant only and may prove useful in the future. Finally, we summarise this chapter in Section 4.5.

https://doi.org/10.24355/dbbs.084-202002271120-0

4.1 Deriving Feature Mappings For Semantic Edits

Depending on the software development approach and the chosen representation of feature mappings, they can be specified at different points in time. For product-line engineering, features are explicitly or implicitly mapped to software artefacts right from the beginning of software development. In approaches, where features are identified later as first-class artefacts to develop, such mappings have to be recovered retroactively in a development-freezing migration phase.

We support ongoing software development of clone-and-own systems up to software product lines as developers can specify feature mappings during their usual programming activity. Opposed to other approaches, such as Ecco [LLHE17, FLLHE15] and VTS [SBWW16], software artefacts are mapped to features or their interactions directly while editing them instead when committing changes to variation or version control systems. Thereby, we enable flexibly changing the feature context during development in contrast to the transaction based VTS and ECCO. To specify a feature context, edits have to be made in an IDE. However, we still support development outside of an IDE when no feature context has to be specified while preserving and incorporating existing feature mappings. Thus, we do not alter developer's accustomed versioning workflow but enhance text editing by optional feature mapping specification possibilities that are comparably laborious to writing preprocessor macros.

In the following, we first introduce our algorithm for semi-automated feature mapping derivation during software development. Second, we depict general constraints that apply to all feature mappings. Third, we discuss why the absence feature mappings or the feature context is an issue and how it can be interpreted. Afterwards, we introduce our typed functions for deriving feature mappings upon edits: $\mathscr{F}_{\text{insert}}$, $\mathscr{F}_{\text{delete}}$, $\mathscr{F}_{\text{update}}$, and $\mathscr{F}_{\text{move}}$. We present each edit operation in its own section in which we also clarify which variants need to be synchronised and how.

4.1.1 Feature Mapping Derivation Algorithm

To incorporate the domain knowledge of developers during editing, we introduce the *feature context*, a propositional formula over the set of features. Developers depict the feature or feature interaction they are currently working on by specifying the corresponding feature context explicitly. We can use the feature context to derive the mapping of the currently edited artefact, as we represent feature mappings as propositional formulas, too. To derive the feature mapping from one artefact's version to another, each edit is associated to a feature context:

Definition 4.1 (Feature Context)**.** *For an edit script* $\mathscr{E} = (e_1, \ldots, e_n)$*, let* $\mathcal{FC}(e_i) \in \mathcal{B}(\mathrm{F}) \cup \{\text{null}\}$ *denote the feature context over features* F *assigned to each semantic edit* e_i *with* $i \in \{1, \ldots, n\}$*.*

To edits outside of an IDE and to not enforce domain knowledge specification (e.g., when it is not present), the feature context is deliberately allowed to be unspecified (i.e., set to *null*).

https://doi.org/10.24355/dbbs.084-202002271120-0

To make use of feature mappings for targeted synchronisation between variants, each variant is identified by the set of features it implements, its *configuration*:

Definition 4.2 (Variants). *Let $\mathcal{V} \subseteq \mathcal{P}(\mathcal{A})$ denote the software variants of a clone-and-own project, where $\mathcal{P}$ denotes the power set as defined in Section 2.3.*

Definition 4.3 (Configurations). *Let $\mathrm{C} : \mathcal{V} \mapsto \mathcal{P}(F)$ be the configuration, i.e., the set of chosen features, of a given variant.*

Correspondingly, $\mathrm{F} \setminus \mathrm{C}(V)$ contains all features that are deselected in variant V. Thus, configurations of variants are not partial, i.e., the selection state of all features is known and given by their in- or exclusion in / from the configuration.

As explained in the previous chapter, we assign features to nodes of ASTs. To guarantee disciplined annotations, i.e., syntactical correctness upon arbitrary feature composition and removal, only *syntactically optional* nodes, defined in Definition 3.1, can be mapped to features. We refer to such nodes as *feature mapping fit* denoted by the predicate Fit introduced in Definition 3.8. To be able to express feature interactions, we allow any propositional formula for feature mappings. As artefacts can be unmapped, especially at the beginning of development, we also allow the absence of a mapping, represented by the value *null*, as a possible value:

Definition 4.4 (Feature Mapping). *For a set of features F, let $\mathcal{F} : \Cap \to \mathcal{B}(F) \cup \{\mathrm{null}\}$ be the feature mapping of an AST node. Only nodes that are feature mapping fit can be assigned a feature mapping, i.e., $\neg\,\mathrm{Fit}(a) \Rightarrow \mathcal{F}(a) = \mathrm{null}$.*

For each implementation artefact $a \in \mathcal{A}$, its mapping is given by its corresponding AST node. For practical reasons, the domain of function $\mathcal{F}$ should be restricted to the artefacts present in the variants $\mathcal{V}$.

To model hierarchical dependencies between implementation artefacts, such as containment of method definitions inside a class, AST nodes propagate their feature mapping if they have one. Thus, any *hierarchically mandatory* node (see Definition 3.2 on Page 21) propagates its mapping as it is a mandatory element in the hierarchy for its descendants. This gives us the actual presence condition of an artefact:

Definition 4.5 (Propagated Feature Mapping). *For an AST T and set of features F, let $\mathcal{F}^* : \Cap \to \mathcal{B}(F) \cup \{\mathrm{null}\}$ be the presence condition of an AST node, i.e., the mapping that is propagated to it by its ancestors.*

Formally, $\mathcal{F}^*(a) \coloneqq \mathcal{F}(a) \wedge \mathcal{F}^*(\widehat{parent}_T(a))$ with $\mathcal{F}^*(root(T)) \coloneqq \mathcal{F}(root(T))$ and $\mathcal{F}^*(\varepsilon) \coloneqq \mathcal{F}(\varepsilon) \coloneqq \mathrm{true}$ (cf. Definition 4.5 on Page 39).

As defined in Section 2.3, the propositional value *null* is treated as the neutral element in operations. Thus, if a node a does not have a mapping (i.e., $\mathcal{F}(a) = null$), it neither invalidates nor contributes to its presence condition because $\mathcal{F}(a) \wedge \mathcal{F}^*(parent_T(a)) = null \wedge \mathcal{F}^*(parent_T(a)) = \mathcal{F}^*(parent_T(a))$. For the same reason, a node mapped to null *null* also does not influence the presence conditions of its descendants.

https://doi.org/10.24355/dbbs.084-202002271120-0

When editing an unmapped artefact under a specific feature context φ, one would expect that artefact to be mapped to φ afterwards. However, a distinction between the kind of change performed has to be made because a deletion under φ likely indicates that this deletion should be performed in all variants satisfying φ. Thus, the deleted artefact is not part of those variants anymore and has to be mapped to $\neg\,\varphi$ as it can only be present in variants not satisfying φ afterwards.

Our approach for semi-automatic feature mapping derivation takes the old and new version of an implementation artefact $a_{old},\ a_{new} \in \mathcal{A}$ together with the feature mapping $\mathcal{F}$ effective before the edit and feature context $\mathcal{FC}$ specified by the user to compute the new mapping $\mathcal{F}'$:

$$\mathscr{F}(a_{old}, a_{new}, \mathcal{F}, \mathcal{FC}) = \mathcal{F}' \,. \tag{4.1}$$

Our feature mapping derivation is defined in Algorithm 4.1. In the beginning, it parses both artefact versions a_{old} and a_{new} to ASTs to compute an initial edit script $\mathscr{E}$ in Line 3, transforming the T_{old} to T_{new}. Next, we derive a semantic edit script $\hat{\mathscr{E}}$ by lifting $\mathscr{E}$ to user-level edits. In the fifth step, we lift the raw feature contexts $\mathcal{FC}$ specified by the developer during changing a_{old} to a_{new} to the semantic edits. Therefore, we have to find a way for sophisticated feature context recording such that feature contexts can be mapped to semantic edits in a reasonable way. We investigate on function lift in Chapter 5. It is important to consider that the feature context $\mathcal{FC}(\hat{e}_i)$ of a semantic edit $\hat{e}_i$ can be undefined, i.e., $\mathcal{FC}(\hat{e}_i) = \mathit{null}$. Beginning with Line 8, we iteratively deduce the new mapping for each single semantic edit. By applying $\hat{e}_i$ to the tree T_{i-1} in Line 9, we obtain the next tree $\hat{e}_i(T_{i-1}) = T_i$, such that in the end T_n is equal to the AST T_{new} of the new artefact a_{new}. With four dedicated derivation functions $\mathscr{F}_t$ each edit is handled corresponding to its type $t \in \{\text{insert, delete, update, move}\}$. Each derivation function $\mathscr{F}_{\text{insert}}$, $\mathscr{F}_{\text{delete}}$, $\mathscr{F}_{\text{update}}$, and $\mathscr{F}_{\text{move}}$ takes

- the AST T_{i-1} before the edit,
- the AST T_i after the edit,
- edited AST nodes $A \subseteq T_{i-1} \cup T_i$,
- old feature mapping $\mathcal{F}'$ defined on T_{i-1},
- and feature context $\mathcal{FC}(\hat{e}_i)$

to derive the next feature mapping for T_i. Here, the semantic edit $\hat{e}_i$ itself needs not to be considered in detail anymore as we handled its type by choosing a specialised derivation function $\mathscr{F}_t$ and collecting the involved nodes in the set A. The set of involved nodes A depending on the edit type is constructed as follows:

Insertion $A \coloneqq T_i \setminus T_{i-1}$ contains exactly the nodes inserted to the tree of which none was present before the edit.

Deletion $A \coloneqq T_{i-1} \setminus T_i$ contains exactly the nodes removed from the tree of which none is present after the edit, i.e., in the new tree.

https://doi.org/10.24355/dbbs.084-202002271120-0

Algorithm 4.1 Feature Mapping Derivation $\mathscr{F}$

Input: $a_{old}, a_{new} \in \mathcal{A}$; old feature mapping $\mathcal{F}$; feature context $\mathcal{FC}$
Output: $\mathcal{F}'$

1: $T_{old} \leftarrow \text{parseAST}(a_{old})$
2: $T_{new} \leftarrow \text{parseAST}(a_{new})$
3: $\mathscr{E} \leftarrow \text{computeEditScript}(T_{old}, T_{new})$
4: $\hat{\mathscr{E}} = (\hat{e_1}, \ldots, \hat{e_n}) \leftarrow \text{semanticLifting}(T_{old}, \mathscr{E})$
5: $\mathcal{FC} \leftarrow \text{lift}(\mathcal{FC}, \hat{\mathscr{E}})$

6: $T_0 \leftarrow T_{old}$
7: $\mathcal{F}' \leftarrow \mathcal{F}$
8: **for** $i \in \{1, \ldots, n\}$ **do**
9: $\quad T_i \leftarrow \hat{e}_i(T_{i-1})$
10: $\quad A \leftarrow$ AST nodes involved in $\hat{e}_i$
11: $\quad t \leftarrow$ type of $\hat{e}_i \in \{\text{insert, delete, update, move}\}$
12: $\quad \mathcal{F}' \leftarrow \mathscr{F}_t(T_{i-1}, T_i, A, \mathcal{F}', \mathcal{FC}(\hat{e}_i))$
13: **end for**

14: **assert** $T_n = T_{new}$

15: **return** $\mathcal{F}'$

Update A contains all nodes whose type or value has changed.

Move To conform to any move operation, A contains the nodes of the partial ASTs that got moved.

Due to the feature mapping propagation throughout the AST, feature mappings are adjusted for syntactical correctness implicitly, for instance when moving a method to another class. Therefore, our edit operations do not have to decompose existent mappings to reassemble them because hierarchical inclusion is guaranteed automatically. This is possible because the actual presence condition $\mathcal{F}^*(a)$ is decoupled from the feature mappings $\mathcal{F}(a)$ of single nodes a. Thus, each edit operation can focus on deriving knowledge of mappings of single artefacts by incorporating the feature context.

Our algorithm is independent from concrete tree differencing algorithms and semantic lifting implementations, used in Line 3 and 4. It only relies on the classification of edits as either insertion, deletion, update, or move, and on the involved nodes A. Nevertheless, even tree differencing algorithms not supporting certain edit operations (e.g., RTED that does not consider moves as summarised in Table 3.2), can be used with condoning a loss in accuracy. Thus, our algorithm can reuse any existing tree differencing algorithm for low-level edit scripts. Semantic lifting is optional as it only refines the edit script.

https://doi.org/10.24355/dbbs.084-202002271120-0

4.1.2 Constraints on Feature Mappings

Independently of how feature mappings are specified or (semi-) automatically derived, some constraints apply to all of them. These prevent malformed or contradicting feature mappings leading to defective variants or inconsistent behaviour, e.g., bugs. Thus, for any feature mapping and feature context over a set of features, the following constraints have to be met:

Intra Variant Compliance – Each variant V is identified by its configuration $\mathrm{C}(V)$ that describes which features are implemented in V and which are not. Thus, if an artefact is present in a variant it belongs to features selected in that variant's configuration.[1] Therefore, the configurations $\mathrm{C}(V)$ of all variants V containing a (i.e., $a \in V$) have to satisfy its presence condition $\mathcal{F}^*(a)$. Otherwise, a could not be part of those variants:

$$\models \forall V \in \mathcal{V} \,.\, \forall a \in V.\ \mathrm{SAT}(\mathcal{F}^*(a) \wedge \bigwedge_{f \in \mathrm{C}(V)} f \wedge \bigwedge_{f \in \mathrm{F} \setminus \mathrm{C}(V)} \neg f).$$

As we identify assignments to propositional formulas as sets of variables (as defined in Section 2.3), configurations themselves can also be seen as assignments of features. In that sense, they assign each feature f the value *true* or *false* depending on its presence in said configuration, i.e., $f \in \mathrm{C}(V)$ for a variant V. Thus, $\mathcal{F}^*(a)$ has to evaluate to *true* under the assignment $\mathrm{C}(V)$ and no real satisfiability checks are necessary. We write:

$$\models \forall V \in \mathcal{V} \,.\, \forall a \in V.\ \mathit{eval}(\mathrm{C}(V), \mathcal{F}^*(a)). \tag{4.2}$$

Furthermore, feature mappings should comply the global feature model. Per assumption, configurations of variants are already created according to the global feature model as stated in Section 3.1. All feature mappings satisfied by a variant's configuration thereby are implicitly conforming the feature model.

Inter Variant Compliance – Our goal is the incremental synchronisation between variants. During clone-and-own development, variants deliberately must not be synchronised at all. Hence, we do only (and we can only) enforce variant synchronisation for already mapped artefacts. Thus, for each artefact $a \in \mathcal{A}$ having a feature mapping (i.e. $\mathcal{F}^*(a) \neq \mathit{null}$) the variants satisfying its mapping also contain a:

$$\forall a \in \mathcal{A} \,.\ \mathcal{F}^*(a) \neq \mathit{null} \models \ \big(\forall V \in \mathcal{V} \,.\, \mathit{eval}(\mathrm{C}(V), \mathcal{F}^*(a)) \Rightarrow a \in V\big). \tag{4.3}$$

Thus, our constraint enforces a consistent implementation of features across variants step by step.

Both constraints are required for consistent and valid feature mappings across variants. Which variants need to be synchronised depends on the type of edit and our newly derived feature mapping. If an artefact is assigned a new feature mapping, it should be contained in all variants satisfying this new mapping. Vice versa, it

[1] Technically, an artefact can also belong to an unselected feature if its mapping is negative.

https://doi.org/10.24355/dbbs.084-202002271120-0

should not be contained in variants anymore satisfying the old mapping but not the new one. Thus, if a variant satisfied the old mapping but not the new mapping, the corresponding artefact needs to be deleted from it.

For each type of edit we cover in detail which variants need to be synchronised and how. Therefore, we do not have to consider intra and *inter variant compliance* for our following derivation of feature mapping as both constraints are resolved during the subsequent synchronisation step. Nevertheless, as feature mappings identify target variants to synchronise we derive new feature mappings such that they conform to both constraints in a reasonable and meaningful way.

4.1.3 Interpretation of Absent Feature Mappings

As we notably support ongoing development on previously unmanaged clone-and-own software, artefacts can be unmapped. Such null-mappings represent the absence of a feature mapping and contain no information. Unmapped artefacts can be present in any variant and initially, all artefacts are unmapped. We denote the null-mapping with *null* as usual in programming languages.

It is important to consider, how such absent or null-mappings should be interpreted. We identified two possible interpretations for an empty feature context $\mathcal{FC}(e) = null$ for an edit e: *don't know* and *don't care*.

The *don't know* interpretation conforms the case of developers not knowing on which feature they are working on. In Section 3.1, we assumed that developers typically know on which feature they are working. As a fallback however, we support the rarer case of developers not knowing the feature they are working on. For instance, if developers have to change roles such that they have to work on source code they do not know, they might not be able to specify the feature context. If we interpret the absence of a feature context as *don't know*, derived mappings of edited artefacts become uncertain because we do not know how the edit affects existing mappings. This could even require to erase (i.e., set to *null*) existing mappings when mapped artefacts are edited because we do not know if the edited artefact still belongs to its former feature.

In contrast, the *don't care* interpretation states that developers do not care to which feature the currently edited artefact belongs to. Again, developers do not provide any domain knowledge but ensure to not invalidate existing mappings. For instance, developers could omit the feature context when an artefact is already mapped, i.e., the feature context would be the same as the mapping of the edited artefact. Thus, we can and have to use existing mappings for feature mapping derivation.

We stick to the *don't care* interpretation as our main goal is the successive synchronisation between variants. By keeping existing mappings as is, we avoid the introduction of uncertainties and inconsistencies opposed to the *don't know* interpretation which may erase existent mappings in the edited variant due to uncertainty on edits. Even if developers do not know on which feature they are working on, existing mappings may still remain valid as other developers specified them so earlier.

https://doi.org/10.24355/dbbs.084-202002271120-0

4.1.4 Deriving Feature Mappings Upon Insertions

In this section, we formally define our feature mapping derivation function $\mathscr{F}_{\text{insert}}$ for artefact insertions. Therefore, we consider the feature context φ, old AST T_{i-1}, new AST T_i, and the set of inserted nodes A, as defined in Section 4.1. For our derivation of feature mappings it is not important how exactly elements got inserted, as long as edits were considered as an insertion during the semantic lifting phase depicted in Chapter 3. However, we make one assumption for insertions: Whenever the parent of an already existing node changes, that parent is one of the inserted nodes and the previous parent is still an ancestor:

$$\begin{aligned} \forall a \in T_{i-1} : \quad & \mathit{parent}_{T_{i-1}}(a) \neq \mathit{parent}_{T_i}(a) \\ & \Rightarrow \mathit{parent}_{T_i}(a) \in A \wedge a \in \mathit{child}^*_{T_i}(\mathit{parent}_{T_{i-1}}(a)). \end{aligned} \tag{4.4}$$

In particular, this means that existing nodes cannot be relocated arbitrarily upon insertions but only below new nodes. This constraints also covers changes to the root of a tree. If a node $a \in T_{i-1}$ was the root but is no more (i.e., $\mathit{root}(T_{i-1}) = a \neq \mathit{root}(T_i)$), then its new parent has to be one of the new nodes. As desired, the other case of $a \in T_{i-1}$ becoming the root (i.e., $\mathit{root}(T_{i-1}) \neq a = \mathit{root}(T_i)$), is implicitly prohibited because its new parent would have to be an inserted node but a does not have a parent.

First, we have to warrant inserted artefacts to be mapped such that their presence in the edited variant is ensured. Otherwise, the new mapping could violate the configuration of the edited variant. This could even result in the necessity to remove the currently edited artefact although it was just inserted (or affected by the insertion of another artefact). Considering the feature context φ describing the feature or feature interaction developers are working on, the inserted artefact should be present in all variants satisfying φ. In particular, the currently edit variant satisfies φ as we assumed in Section 4.1.2. Therefore, the new feature mapping $\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a)$ of a newly inserted artefact $a \in A$ should be satisfied in all variants in which the feature context φ is satisfied:

$$\forall a \in A.\, \varphi \models \mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a). \tag{4.5}$$

We refer to this constraint as *no guesses* as it preserves a reasonable behaviour by disallowing arbitrarily strong or unrelated feature mappings, such as the trivial mapping $false$. Thereby, *no guesses* ensures the feature mapping being satisfied in a variant whenever the feature context is. Note that we still allow the actual presence condition $\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a)$ to be different, i.e., there may be variants satisfying φ but not the actual presence condition $\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a)$ of an inserted artefact $a \in A$. In that case, a is actually part of a feature interaction which is only partially described partially by φ.

Second, we need to ensure the incorporation of the feature context reflecting the developer's domain knowledge. Whenever the presence condition of a newly inserted artefact is satisfied by a variant's configuration, the feature context should also be satisfied. Otherwise, there may be configurations for which the presence condition is satisfied but not the feature context. Then, variants could be identified as synchronisation targets that are incompatible to the specified feature context. Hence,

https://doi.org/10.24355/dbbs.084-202002271120-0

the presence condition $\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a)$ of the new artefacts A should be as least as strong as the feature context:

$$\forall a \in A.\, \mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) \models \varphi\,. \tag{4.6}$$

We refer to this constraint as *intention insurance* as it ensures the incorporation of the developers intention expressed by the feature context. It disallows arbitrarily weak feature mappings, such as the trivial mapping *true*, saying an artefact is part of every variant, when a feature context is specified. Note that we reason about the actual presence condition $\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*$ rather than just $\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)$ to not enforce unnecessary redundancies when already its *hierarchically mandatory* parent nodes satisfy *intention insurance*.

Third, we have to ensure feature mappings of existing artefacts to remain constant. As insert operations may change the structure of existing nodes, such as the $\mathit{insert}_{\text{partial}}$ edit operation (introduced in Section 3.3.1) does, even old nodes may be affected by insertions. Typically, such changes cannot affect the presence condition of an existing artefact because therefore an *hierarchically mandatory* ancestor would have to be removed or added. Removing an *hierarchically mandatory* parent would invalidate an AST, and adding one would mean that the AST was invalid before. However, they may be special cases where an *hierarchically mandatory* can be added. For instance, a field of a class could be enclosed by a method and turned into a local variable.[2] Another example would be syntactic dependencies that can only be detected with dependency analyses on the AST. For instance, in some programming languages, variables can be defined inside expressions of conditions or loops and be used inside that scopes (e.g., `if (bool b = ...) { print(b); }`). Such a scope may has to propagate its mapping to its children, although those were independent of that scope before. We will investigate further classification of AST nodes in the future and discuss it again in Chapter 9. For now, as we do not want to exclude such cases straight away, we allow the feature context to augment the presence conditions of existing artefacts:

$$\begin{aligned} \models \forall a \in T_{i-1}. \quad & \mathcal{F}(a) = \mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) \\ & \wedge \big(\mathcal{F}^*(a) \wedge \varphi \Rightarrow \mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a)\big) \\ & \wedge \big(\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) \Rightarrow \mathcal{F}^*(a)\big). \end{aligned} \tag{4.7}$$

We refer to this constraint as *inviolacy of the living* as it ensures feature mappings of existing artefacts to remain unchanged and their presence conditions to be changed only by augmenting them with the feature context at most. Sontag et al. already empirically identified that for two formulas $X, Y \in \mathcal{B}$, the constraints $X \wedge Y \models F$ and $F \models Y$ to shrink the space of possible solutions for $F \in \mathcal{B}$ to just two formulas [Son18]. When considering all 16 possible boolean combinations (e.g., $\wedge$, $\vee$, $\oplus$) of X and Y, the constraints restrict those to only $F = Y$ and $F = X \wedge Y$. Every other combination, such as $X \oplus Y$ or $X \vee \neg Y$, violate at least one of the constraints. Applying $X \wedge Y \models F$ to all 16 possible combinations eliminates half of those such that eight possible combinations remain [Son18, p. 27]. Applying

[2] This may require additional type changes of the nodes (e.g., from `Field` to `LocalVariable`) but that depends on the target languages grammar.

https://doi.org/10.24355/dbbs.084-202002271120-0

$F \models B$ to the remaining combinations restricts those to only the two formulas Y and $X \wedge Y$ [Son18, p. 27]. By substituting $X = \varphi$ and $Y = \mathcal{F}^*(a)$ for an artefact $a \in T_{i-1}$, we obtain only two possible values for $\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a)$, namely $\mathcal{F}^*(a)$ and $\mathcal{F}^*(a) \wedge \varphi$ as desired.

By combining *no guesses* (Constraint 4.5) and *intention insurance* (Constraint 4.6), we see that assigning the feature context φ satisfies both constraints. However, to avoid redundancies in the first place, we can just assign the mapping *true* to an inserted node, if it has a propagating (i.e., *hierarchically mandatory*) parent satisfying *intention insurance* already. Furthermore, as existing nodes should not be affected by the insertion, their mapping should not change:

$$\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) \coloneqq \begin{cases} \mathit{true}, & \mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a)) \models \varphi,\ a \in A, \\ \varphi, & \mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a)) \not\models \varphi,\ a \in A, \\ \mathcal{F}(a), & \text{else.} \end{cases} \tag{4.8}$$

In the following, we will show that feature mappings derived with $\mathscr{F}_{\text{insert}}$ satisfy our constraints *no guesses*, *intention insurance*, and *inviolacy of the living*.

Theorem 4.1. *Feature mappings derived upon insertions with $\mathscr{F}_{\text{insert}}$ satisfy both, no guesses and intention insurance.*

Proof. As both constraints apply to newly inserted nodes only, let $a \in A$ be an arbitrary but fixed node inserted to the tree. By definition, there are two possible values for a derived by $\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)$:

1. If $\mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a)) \models \varphi$, the node a will be mapped to *true*. As $\varphi \models \mathit{true}$ is a tautology for any feature context φ, *no guesses* is satisfied. Furthermore,

$$\begin{aligned} \mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) &= \mathit{true} \wedge \mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a)) \\ &= \mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a)) \\ &\models \varphi\,. \end{aligned}$$

   Thus, *intention insurance* is satisfied.

2. If $\mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a)) \not\models \varphi$, the node a will be mapped to φ. As $\varphi \models \varphi$ is a tautology for any feature context φ, *no guesses* is satisfied. Furthermore,

$$\begin{aligned} \mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) &= \varphi \wedge \mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a)) \\ &\models \varphi\,. \end{aligned}$$

   Thus, *intention insurance* is satisfied.

In all possible cases (1 and 2), both constraints are satisfied. As $a \in A$ was chosen arbitrarily, both constraints are satisfied for all inserted nodes $a \in A$. Thus *no guesses* and *intention insurance* are satisfied for feature mappings derived with $\mathscr{F}_{\text{insert}}$. □

https://doi.org/10.24355/dbbs.084-202002271120-0

Theorem 4.2. *Feature mappings derived upon insertions with $\mathscr{F}_{\text{insert}}$ satisfy inviolacy of the living.*

Proof. Let $a \in T_{i-1}$ be an arbitrary but fixed node present before the insertion. By definition:

$$\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) = \mathcal{F}(a). \qquad \text{(referred to as (*))}$$

Thus, the explicit mapping of a does not change. The presence condition of a may or may not change:

1. $\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) = \mathcal{F}^*(a)$:
   Then
   $$\begin{aligned} &\mathcal{F}^*(a) \wedge \varphi \Rightarrow \mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) \\ \overset{(1)}{\equiv}\ &\mathcal{F}^*(a) \wedge \varphi \Rightarrow \mathcal{F}^*(a) \qquad \text{(referred to as (**))} \end{aligned}$$
   is a tautology. Further
   $$\begin{aligned} &\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) \Rightarrow \mathcal{F}^*(a) \\ \overset{(*)}{\equiv}\ &\mathcal{F}(a) \Rightarrow \mathcal{F}^*(a) \qquad \text{(referred to as (***))} \end{aligned}$$
   is also a tautology. By combining (*), (**), and (***), we see that *inviolacy of the living* is satisfied.

2. $\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) \neq \mathcal{F}^*(a)$:
   The presence condition of a can be changed in four ways:

   (i) The feature mapping of a changes, i.e., $\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) \neq \mathcal{F}(a)$. This is not the case because of (*).

   (ii) The feature mapping of a propagating ancestor changes, i.e., there is at least one node $p \in T_{i-1} \cap T_i$ with Propagates$(p) = \mathit{true}$, $a \in \mathit{child}^*_{T_i}(p)$, and $\mathcal{F}(p) \neq \mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(p)$. By definition of $\mathscr{F}_{\text{insert}}$, mappings do change only for nodes in A, i.e.,
   $$\mathcal{F}(p) \neq \mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(p) \Leftrightarrow p \in A.$$
   This is a contradiction to $p \in T_{i-1}$. Hence, this case can also not occur.

   (iii) One or more new propagating nodes $P \subseteq T_i \setminus T_{i-1}, P \neq \emptyset$ became ancestors of a. Let $\mathit{ancestors}_{\text{T}}(n) = \{\alpha \in T \mid n \in \mathit{child}^*_T(\alpha)\}$ be the ancestors of a node n in tree T. Because of Assumption 4.4, the previous parents of the children of the nodes P are still ancestors of those children and thereby ancestors of a. Hence, we know that all ancestors of a in the old tree T_{i-1} are still present in the new tree T_i, i.e. $\mathit{ancestors}_{\text{T}_\text{i}}(a) = \mathit{ancestors}_{\text{T}_\text{i-1}}(a) \dot\cup P$. Therefore,
   $$\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) = \mathcal{F}^*(a) \wedge \bigwedge_{p \in P} \mathcal{F}(p).$$

https://doi.org/10.24355/dbbs.084-202002271120-0

By Assumption 4.4, $P \subset A$ because all children of nodes P got their parent changed as $P \subseteq T_i \setminus T_{i-1}$. Thus, $\forall p \in P$ we know

$$\begin{aligned} &\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(p) = \mathit{true} \\ \text{or } &\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(p) = \varphi\,. \end{aligned}$$

And thereby,

$$\begin{aligned} &\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) = \mathcal{F}^*(a) \\ \text{or } &\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) = \mathcal{F}^*(a) \wedge \varphi\,. \end{aligned}$$

Both cases satisfy

$$\begin{aligned} &\mathcal{F}^*(a) \wedge \varphi \models \mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) \\ \text{and } &\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) \models \mathcal{F}^*(a). \end{aligned}$$

Combined with (*), we see that *inviolacy of the living* is satisfied.

(iv) One or more propagating ancestors $P \subseteq T_{i-1} \setminus T_i$ of a got removed. Let $p \in P$ an arbitrary but fixed of those removed nodes and $c \in T_{i-1}$ be an arbitrary but fixed child of p, i.e., $\mathit{parent}_{T_{i-1}}(c) = p$. As p is not present in the new tree T_i anymore and $p = \mathit{parent}_{T_{i-1}}(c)$,

$$\mathit{parent}_{T_{i-1}}(c) \neq \mathit{parent}_{T_i}(c).$$

By Assumption 4.4, we know then

$$\begin{aligned} &c \in \mathit{child}^*_{T_i}(\mathit{parent}_{T_{i-1}}(c)) \\ \equiv\ &c \in \mathit{child}^*_{T_i}(p). \end{aligned}$$

This, however, is a contradiction to $p \in P$ because $P \subseteq T_{i-1} \setminus T_i$ and thereby $p \notin T_i$. As p and c were chosen arbitrarily, there cannot be propagating ancestors of a being removed.

As the new presence condition $\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a)$ of a satisfies *inviolacy of the living* for all possible cases (i.e., for 2iii), $\mathscr{F}_{\text{insert}}$ satisfies *inviolacy of the living* when $\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) \neq \mathcal{F}^*(a)$.

As $\mathscr{F}_{\text{insert}}$ satisfies *inviolacy of the living* for an arbitrarily but fixed chosen $a \in T_{i-1}$ in both cases (1 and 2), it satisfies *inviolacy of the living*. □

We now can derive feature mappings upon insertion of artefacts. We proved that our derivation $\mathscr{F}_{\text{insert}}$ behaves as desired: It incorporates the feature context, such that variants are identified as synchronisation targets in a reasonable way, and it does not infer unrelated features to presence conditions. Moreover, the feature mappings of already existing nodes do not change and their presence conditions can only be augmented by the feature context (if a new ancestor is inserted).

https://doi.org/10.24355/dbbs.084-202002271120-0

4.1.5 Deriving Feature Mappings Upon Deletions

In our enhanced clone-and-own scenario, the deletion of an artefact $a \in A$ should be propagated to all variants that are not supposed to contain a anymore. Therefore, we have to derive a new feature mapping $\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a)$ identifying those variants. Depending on the presence of a mapping $\mathcal{F}^*(a)$ for a before the deletion (i.e., $\mathcal{F}^*(a) \neq \mathit{null}$) and the presence of a feature context (i.e., $\varphi \neq \mathit{null}$) a new feature mapping can be derived with different meaning. Although removed artefacts are not present in the resulting tree T_i, we still assign feature mappings to them for change synchronisation and to introduce the mapping to variants in which the artefact might not get deleted.

If deleted artefacts $a \in A$ do not have an explicit mapping (i.e., $\mathcal{F}(a) = \mathit{null}$), they could still have a presence condition $\mathcal{F}^*(a)$. Depending on the feature context, removing an unmapped method from a mapped class could mean that this method has to be deleted from all variants containing that class. Hence, we always consider the actual presence condition of artefacts for mapping derivation, to identify synchronisation targets correctly.

Not only the old feature mapping of a deleted artefact but also the feature context can be undefined. Thus, we distinguish between its absence $\varphi = \mathit{null}$ and its presence $\varphi \neq \mathit{null}$.

Deletions Without Feature Context

As defined earlier, we stick to the *don't care* interpretation for an empty feature context. In that sense, if no feature context is specified, we propagate the deletion of an artefact to all other variants containing it, i.e., satisfying its presence condition.

If both, the feature context is absent and the deleted artefact $a \in A$ does not have a former mapping (i.e., $\mathcal{F}^*(a) = \mathit{null}$), the developer has not provided any domain knowledge to us. Nevertheless, before the deletion, a was present in the edited variant. Thus, its variant's configuration had to be a satisfying assignment for its unknown mapping $\mathcal{F}(a)$. Otherwise, a could not have been part of its variant V. So, although we do not know $\mathcal{F}(a)$, we do know that its variant's configuration satisfies $\mathcal{F}(a)$. Contrary, a is not present in the current variant after the deletion. Therefore, its new feature mapping is not allowed to be satisfied because a would be part of that variant otherwise. Using the knowledge of configurations of variants this way, could be used to find assignments for $\mathcal{F}(a)$ that either satisfy it or do not. These assignments could be used to synthesise a partial mapping for a, even though no domain knowledge was provided. We elaborate further on this in Section 4.3 on Page 63.

For now, we do not have any knowledge about the feature mapping of a as the developer did not specify a feature context. Thus, when the existing mapping $\mathcal{F}^*(a)$ as well as the feature context φ are undefined, we do not derive any mapping:

$$\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) \coloneqq \mathit{null}, \qquad \text{if } \varphi = \mathit{null} = \mathcal{F}^*(a). \tag{4.9}$$

We could assign the complete negated current variant's configuration as the new feature mapping, but it would be too specific as it does not identify other possible

https://doi.org/10.24355/dbbs.084-202002271120-0

target variants. Besides its technical impracticality, we suspect such huge feature mapping formulas to be unintuitive and counter-productive.

As we assume configurations of variants to be constant as stated in Section 3.1, entire features themselves cannot be added to or removed from variants.[3] Thus, deleting an artefact $a \in A$ cannot indicate the deletion of the entire feature $\mathcal{F}^*(a)$ which would result in a configuration change. Hence, if $a \in A$ has a presence condition (i.e. $\mathcal{F}^*(a) \neq \mathit{null}$), removing a must mean that a does not belong to the feature $\mathcal{F}^*(a)$ anymore. So, its new presence condition $\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a)$ cannot be satisfied in variants containing a hitherto:

$$\forall a \in A.\ \mathcal{F}^*(a) \neq \mathit{null} \models \big(\mathcal{F}^*(a) \Rightarrow \neg\, \mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a)\big). \tag{4.10}$$

Thus, for any deleted node with a feature mapping, its mapping cannot stay the same. We refer to this constraint as *configuration sentinel* as it ensures that configurations are constant, i.e., deleting an artefact cannot indicate a configuration change but a change in the implementation of a feature. If a gets reinserted, *configuration sentinel* must not hold, as a is present again in the current variant. However, such an operation would be identified as a move operation during semantic lifting instead of a deletion. Therefore, we assume *configuration sentinel* to be satisfied.

When a feature mapping is present, $\mathcal{F}^*(a) \neq \mathit{null}$, we assume that it is already synchronised across variants because of our *inter variant compliance* constraint described in Section 4.1.2 on Page 42. According to our *configuration sentinel* constraint, deleting the mapped artefact a means, that it does not belong to its former feature mapping $\mathcal{F}^*(a)$ anymore. To synchronise all existing implementations of $\mathcal{F}^*(a)$ across variants, a has to be deleted from all implementations of $\mathcal{F}^*(a)$, i.e. exactly from those variants whose configuration satisfies $\mathcal{F}^*(a)$. Furthermore, as all variants satisfying $\mathcal{F}^*(a)$ also contain a, due to our *inter variant compliance* constraint again, the remaining variants not satisfying $\mathcal{F}^*(a)$ do not contain a before the edit. Certainly, these variants should contain a after the edit neither:

$$\forall a \in A.\ \mathcal{F}^*(a) \neq \mathit{null} \models \big(\neg\, \mathcal{F}^*(a) \Rightarrow \neg\, \mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a)\big). \tag{4.11}$$

We refer to this constraint as *no spawn* because it ensures that deleted artefacts will not be inserted into other variants if the developer did not specify so explicitly. In that case, however, the developer would have specified a feature context.

By combining the constraints *configuration sentinel* (Constraint 4.10) and *no spawn* (Constraint 4.11) we obtain:

$$\forall a \in A.\ \mathcal{F}^*(a) \neq \mathit{null} \models \neg\, \mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a). \tag{4.12}$$

Thus, for any assignment, the new presence condition $\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a)$ has to evaluate to *false*. The only propositional formula that evaluates to *false* for any assignment of variables is *false* itself. Hence, we define:

$$\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) \coloneqq \mathit{false}, \qquad \text{if } \varphi = \mathit{null} \neq \mathcal{F}^*(a). \tag{4.13}$$

[3] Instead of changing a configuration, we would create a new variant with the new configuration. This new variant however could then just be cloned from the initial variant whose configuration should be changed.

https://doi.org/10.24355/dbbs.084-202002271120-0

```
generalcase();
```

(a) Initial Version

```
#ifdef FANCY
  fancycase();
#else
  generalcase();
#endif
```

(b) Revised Version With Specialised Handling for feature `FANCY`

Figure 4.1: Deletion of Unmapped Artefacts Under Feature Context FANCY in Software Product-Line Engineering: The code on the left is changed to the right version. We investigate how such a change would play out in clone-and-own development.

We can still identify the variants to which the deletion should be propagated. These are exactly those variants $V_{target} \in \mathcal{V}$ containing a before the deletion, i.e., for which $eval(\mathrm{C}(V_{target}), \mathcal{F}^*(a)) = true$ but do not so afterwards, i.e., $eval(\mathrm{C}(V_{target}), \mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a)) = false$, as no variant can.

Deletions With Feature Context

With the feature context φ developers specify the feature or feature interaction they are currently working on. When they delete an artefact $a \in A$, they indicate that a does not belong to that feature context anymore. Then, a has to be deleted in any variant satisfying φ. Hence, a needs a presence condition that ensures its absence in any variant satisfying φ:

$$\forall a \in A.\ \mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) \models \neg\, \varphi\,. \tag{4.14}$$

We refer to this constraint as *intention insurance* as for its counterpart for insertions because it serves the same purpose of ensuring the incorporation of the feature context.

As opposed to the already synchronised state described in the previous subsection, here a should not be deleted in all variants but instead may deliberately be kept:

Example 4.1.1. Consider Figure 4.1. It shows the introduction of new code and feature mappings in a preprocessor-based product line. Thereby, the unmapped code (a) on the left side is changed and annotated leading to the code fragment (b) on the right side. In our clone-and-own scenario, introducing such a special case in a variant containing the feature `FANCY` would require to first delete the `generalcase()` call, and second to insert the statement `fancycase()` under feature context `FANCY`. Important to notice is that the statement `generalcase()` is not supposed to get deleted in all variants but instead only in those not containing feature `FANCY`. Thereby, the original statement `generalcase()` gets a new mapping, namely ¬`FANCY`, as shown by the `#else` branch in Figure 4.1b. This example also shows the necessity of specifying the feature context `FANCY` not only before inserting `fancycase()` but already upon deletion of `generalcase()`. (Detecting replacements as such with missing feature contexts will be discussed in Chapter 9 on Page 109.)

https://doi.org/10.24355/dbbs.084-202002271120-0

```
#ifdef ORDINARY
  generalcase();
#endif
```

(a) Initial Version

```
#ifdef FANCY
  fancycase();
#elif ORDINARY
  generalcase();
#endif
```

(b) Revised Version With Specialised Handling for feature `FANCY`

Figure 4.2: Deletion of Mapped Artefacts Under Feature Context FANCY in Software Product-Line Engineering: The code on the left is changed to the right version and is similar to that shown in Figure 4.1 but this time the original code on the left side already has a feature mapping.

Although Constraint 4.14 does not enforce $\neg\varphi$ as the direct mapping of the deleted artefact $a \in A$, Example 4.1.1 frugally illustrates its reasonability. By incorporating the presence of a in other variants and their satisfaction of $\neg\varphi$, we could possibly derive more knowledge on the actual mapping of a. We discuss on this possibility in Section 4.3. For now, if no original feature mapping is present for deleted artefacts $a \in A$, we assign the negated feature context to them:

$$\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) := \neg\varphi, \qquad \text{if } \varphi \neq \mathit{null} = \mathcal{F}^*(a). \tag{4.15}$$

As for deletions without feature context, target variants for synchronisation are those variants satisfying the old feature mapping but not the new one. If no old feature mapping is present, these are exactly those variants V satisfying the feature context because

$$\begin{aligned}
&\neg \mathit{eval}(\mathrm{C}(V), \mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a)) \\
\equiv\ &\neg \mathit{eval}(\mathrm{C}(V), \mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) \wedge \mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(\widehat{\mathit{parent}}_T(a)) \\
\equiv\ &\neg \mathit{eval}(\mathrm{C}(V), \neg\varphi \wedge \mathcal{F}^*(\widehat{\mathit{parent}}_T(a)) \\
\equiv\ &\neg \mathit{eval}(\mathrm{C}(V), \neg\varphi \wedge \mathit{null}) \\
\equiv\ &\neg \mathit{eval}(\mathrm{C}(V), \neg\varphi) \\
\equiv\ &\ \mathit{eval}(\mathrm{C}(V), \ \ \varphi).
\end{aligned}$$

Finally, we consider the last possible case, namely the deletion of a mapped artefact $a \in A$ with $\mathcal{F}^*(a) \neq \mathit{null}$ under a given feature context $\varphi \neq \mathit{null}$. Therefore, we consider our previous example again but with the original artefact being mapped:

Example 4.1.2. Figure 4.2 shows the same example as the previous Figure 4.1 but this time, the original code has a feature mapping, namely to feature `ORDINARY`. The example shows that deleting a mapped artefact a must not mean that its already existing mapping $\mathcal{F}(a)$ should be ignored. Instead, by introducing an `#elif` statement, the original mapping is now enhanced to $\neg$`FANCY` $\wedge$ `ORDINARY`. Notice that if no new statement such as `fancycase()` is inserted, deleting a statement still indicates it to not belong to the feature context.

https://doi.org/10.24355/dbbs.084-202002271120-0

The practical Example 4.1.2 depicts that deletions under a feature context do not have to imply the existing feature mapping $\mathcal{F}(a)$ to be negated or ignored. Thus, enhancing a mapping by $\neg\,\varphi$ does not imply the negation of $\mathcal{F}(a)$ as illustrated by the example. Instead, we keep the existing mapping:

$$\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) \coloneqq \mathcal{F}(a) \wedge \neg\,\varphi, \qquad \text{if } \varphi \neq \mathit{null} \neq \mathcal{F}^*(a). \tag{4.16}$$

Bringing it All Together

As all cases of presence and absence of initial feature mapping $\mathcal{F}^*(a)$ and feature context φ are covered, we merge the individual definitions given in Equations 4.9, 4.13, 4.15 and 4.16 to our final function for feature mapping derivation upon deletions:

$$\begin{aligned}
\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) \coloneqq & \begin{cases}
\mathit{null}, & \varphi = \mathit{null} \wedge \mathcal{F}^*(a) = \mathit{null}, a \in A, \\
\mathit{false}, & \varphi = \mathit{null} \wedge \mathcal{F}^*(a) \neq \mathit{null}, a \in A, \\
\neg\,\varphi, & \varphi \neq \mathit{null} \wedge \mathcal{F}^*(a) = \mathit{null}, a \in A, \\
\mathcal{F}(a) \wedge \neg\,\varphi, & \varphi \neq \mathit{null} \wedge \mathcal{F}^*(a) \neq \mathit{null}, a \in A, \\
\mathcal{F}(a), & a \notin A.
\end{cases} \\
= & \begin{cases}
\mathit{false}, & \varphi = \mathit{null} \wedge \mathcal{F}^*(a) \neq \mathit{null}, a \in A, \\
\mathcal{F}(a) \wedge \neg\,\varphi, & \varphi \neq \mathit{null} \vee \mathcal{F}^*(a) = \mathit{null}, a \in A, \\
\mathcal{F}(a), & a \notin A.
\end{cases}
\end{aligned} \tag{4.17}$$

If the feature context φ is the same as the deleted artefact's mapping $\mathcal{F}(a)$, the deleted artefact should be deleted in all variants implementing $\mathcal{F}(a)$ such as for $\varphi = \mathit{null} \wedge \mathcal{F}^*(a) \neq \mathit{null}$. This is indeed the case because $\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) = \mathcal{F}(a) \wedge \neg\,\varphi = \varphi \wedge \neg\,\varphi = \mathit{false}$ for $\varphi = \mathcal{F}(a) \neq \mathit{null}$.

In the following, we prove that $\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)$ meets both constraints, *configuration sentinel* (Constraint 4.10) and *no spawn* (Constraint 4.11), when no feature context is specified. If a feature context is specified, both constraints do not apply.

Theorem 4.3. *$\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*$ conforms to the constraints configuration sentinel and no spawn for any feature context $\varphi \neq$ null.*

Proof. Let $\varphi = \mathit{null}$. We prove the satisfaction of both constraints simultaneously for an arbitrarily deleted artefact $a \in A$. Therefore, it is sufficient to prove the satisfaction of their conjunction given in Constraint 4.12:

$$\mathcal{F}^*(a) \neq \mathit{null} \models \neg\,\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a).$$

https://doi.org/10.24355/dbbs.084-202002271120-0

Let $\mathcal{F}^*(a) \neq \mathit{null}$. As no feature context is specified, $\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) = \mathit{false}$ by definition. Because of Definition 4.5, its propagated mapping is defined as follows:

$$\begin{aligned}\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) &= \mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) \\ &\quad \wedge \mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(\widehat{\mathit{parent}}_{T_i}(a)) \\ &= \mathit{false} \\ &\quad \wedge \mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(\widehat{\mathit{parent}}_{T_i}(a)) \\ &= \mathit{false}\,.\end{aligned}$$

Thus, $\neg\, \mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) = \neg\, \mathit{false} = \mathit{true}$. Thereby, Constraint 4.12 is satisfied. Hence, both, *configuration sentinel* and *no spawn*, are satisfied for any previous feature mapping $\mathcal{F}^*(a) \neq \mathit{null}$ when no feature context is specified. □

In the case of presence of a feature context, *intention insurance* (Constraint 4.14) has to be satisfied:

Theorem 4.4. *$\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*$ conforms to intention insurance for any feature context $\varphi \neq$ null.*

Proof. For any deleted artefact $a \in A$ and feature context $\varphi \neq \mathit{null}$, the derived mapping is defined as $\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) = \mathcal{F}(a) \wedge \neg\, \varphi$. Again, because of Definition 4.5, its propagated mapping is defined as follows:

$$\begin{aligned}\mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) &= \mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) \\ &\quad \wedge \mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(\widehat{\mathit{parent}}_{T_i}(a)) \\ &= \mathcal{F}(a) \wedge \neg\, \varphi \\ &\quad \wedge \mathscr{F}_{\text{delete}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(\widehat{\mathit{parent}}_{T_i}(a)) \\ &\models \neg\, \varphi\end{aligned}$$

Thus, *intention insurance* is satisfied for any deleted artefact $a \in A$. □

We now can derive feature mappings upon deletions such that deletions of artefacts can be synchronised between variants reasonably. Depending on the feature context being specified, developers can either delete artefacts from the entire project or just from a certain set of variants. If a feature context is specified, we proved our derivation $\mathscr{F}_{\text{delete}}$ to ensure deleted artefact to be removed from all variants implementing the feature context.

4.1.6 Deriving Feature Mappings Upon Moves

Depending on the feature context φ and a former mapping $\mathcal{F}$, we derive the new mapping $\mathscr{F}_{\text{move}}$. As for deletions, we consider each case emerging from the presence of an old mapping $\mathcal{F}(a) \stackrel{?}{=} \mathit{null}$ respective the feature context $\varphi \stackrel{?}{=} \mathit{null}$.

Opposed to inserting or deleting, moving artefacts does not change the set of nodes of an AST. Each moved artefact $a \in A$ is present in the currently edited variant

https://doi.org/10.24355/dbbs.084-202002271120-0

$V \in \mathcal{V}$ before and after the edit. As also every other artefact is present in the current variant before and after the edit, new feature mappings of all nodes must be satisfied in the current variant $V \in \mathcal{V}$:

$$\models \forall a \in T_i.\ eval(\mathrm{C}(V), \mathcal{F}^*(a)) \wedge eval(\mathrm{C}(V), \mathscr{F}_{\mathrm{move}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a)). \quad (4.18)$$

To conform to developers' intentions, our derived mapping should be predictable and not produce surprising or confusing results. When developers do not specify a feature context, we have no indication on their intention. We stick to the *don't care* interpretation for absent feature mappings because we do not want to introduce uncertainties and thereby being counterproductive to our goal of synchronising variants successively, as stated in Section 4.1.3 on Page 43. Thus, if a moved artefact has an explicit mapping $\mathcal{F}(a)$ and no feature context is specified, we keep that mapping which also satisfies Constraint 4.18:

$$\forall a \in A.\ \mathscr{F}_{\mathrm{move}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) \coloneqq \mathcal{F}(a), \qquad \text{if } \varphi = null. \quad (4.19)$$

Moving implementation artefacts to other scopes does not violate syntactic constraints because syntactical correctness of mappings is preserved by the AST feature mapping propagation. For instance, the AST propagation automatically produces the syntax preserving presence condition $F_1 \wedge F_2$, when moving a statement s with $\mathcal{F}(s) = F_1$ from one method m_{boring} to another method m_{cool} with $\mathcal{F}(m_{cool}) = F_2$. Note that thereby also inherited mappings of the previous outer scope $\mathcal{F}(m_{boring})$ are removed implicitly from the presence condition of s. Thus, presence conditions are automatically adjusted by our AST propagation when extracting and relocating artefacts.

Moves can not only be performed for single nodes but also for entire subtrees (e.g., as $move_{\mathrm{tree}}$ or $move_{\mathrm{partial}}$ do, defined in Section 3.3.1 on Page 26). The nodes in a subtree may exhibit various different feature mappings. We have no evidence to overwrite these mappings with the feature context φ as they may contain essential interactions. Consider Listing 4.1. Moving the method `ThreadSleep` should not invalidate the mappings of `usleep` and `Sleep`, just as it would not in product-line development. Removing the feature mappings `PAX_OS_LINUX`$\vee$`PAX_OS_ANDROID` and `PAX_OS_WIN` would even destroy the syntactic validity of the program. Thus, only the root node (i.e., `ThreadSleep`) should be augmented with the feature context φ. However, root nodes do not always have to be propagating nodes, i.e., Propagates(r) could be *false* for a root node $r \in A$. To ensure the subtree getting augmented by the feature context, the feature context φ with $\varphi \neq null$ has to be assigned to each moved node $a \in A$ that

- has no moved ancestors that can propagate φ to a:

$$\neg\exists p \in A.\ \mathrm{Propagates}(p) \wedge a \in child^*_{T_i}(p), \quad (4.20)$$

- and does not have a propagating ancestor already satisfying φ (as for $\mathscr{F}_{\mathrm{insert}}$):

$$\mathcal{F}^*(\widehat{parent}_{T_i}(a)) \not\models \varphi\,. \quad (4.21)$$

https://doi.org/10.24355/dbbs.084-202002271120-0

```
1 void ThreadSleep(unsigned int milliseconds) {
2 #if defined(PAX_OS_LINUX) || defined(PAX_OS_ANDROID)
3   usleep(milliseconds * 1000);
4 #elif defined(PAX_OS_WIN)
5   Sleep(milliseconds);
6 #endif
7 }
```

Listing 4.1: Feature Interactions in Preprocessor-Based Software Product Line

Furthermore, unmoved artefacts should not have their feature mapping changed because these artefacts are not edited directly by the developer:

$$\forall a \in T_i \setminus A.\ \mathscr{F}_{\text{move}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) \coloneqq \mathcal{F}(a). \tag{4.22}$$

This does not mean that their presence condition cannot change. As illustrated by the extensively restructurings during $move_{\text{partial}}$, unmoved nodes may be relocated below moved partial subtrees.

Combining Equation 4.19, Constraint 4.20, Constraint 4.21, and Equation 4.22, we obtain our feature mapping derivation upon moves:

$$\mathscr{F}_{\text{move}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) \coloneqq \begin{cases} \mathcal{F}(a) \wedge \varphi, & \varphi \neq \mathit{null}, a \in A, \mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a)) \not\models \varphi, \\ & \text{and } \neg\exists p \in A.\ \text{Propagates}(p) \\ & \qquad \wedge\, a \in \mathit{child}^*_{T_i}(p), \\ \mathcal{F}(a), & \text{else.} \end{cases} \tag{4.23}$$

Notably, $\mathscr{F}_{\text{move}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)$ reasonably produces the same mapping when $\varphi = \mathit{null}$ or $\varphi = \mathcal{F}(a)$.

It is important to consider that while Constraint 4.18 holds for the edited variant, this must not be the case for other variants if new and old mapping differ. When the mapping of a moved artefact changes, there may be variants in which the new mapping is satisfied but the previous mapping was not. Then, the moved artefact needs to be inserted into that variant. Formally, for a moved artefact $a \in A$, consider the following two sets:

$$\mathcal{V}_{ins} = \{V \in \mathcal{V} \mid \neg \mathit{eval}(\mathrm{C}(V), \mathcal{F}^*(a)) \wedge \ \mathit{eval}(\mathrm{C}(V), \mathscr{F}_{\text{move}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a))\},$$
$$\mathcal{V}_{del} = \{V \in \mathcal{V} \mid \ \mathit{eval}(\mathrm{C}(V), \mathcal{F}^*(a)) \wedge \neg \mathit{eval}(\mathrm{C}(V), \mathscr{F}_{\text{move}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a))\}.$$

For both sets of variants either the old or the new mapping is not satisfied. Thus, there are variants in which a move cannot be reproduced as such. Instead, a has to be inserted into the variants $\mathcal{V}_{ins}$ because its new feature mapping is satisfied by their configurations but it is not contained in those variants so far. Contrary, the variants $\mathcal{V}_{del}$ contain a indeed but do not satisfy the new feature mapping anymore. Thus, a has to be deleted from them.

https://doi.org/10.24355/dbbs.084-202002271120-0

4.1.7 Deriving Feature Mappings Upon Updates

Our update allows an AST node to change its name or type. We consider type changes to be rare as they are not considered in the literature as shown in Table 3.2 on Page 33. Usually, type changes are supported by a deletion followed by an insertion, i.e., replacing an existing construct. This is because arbitrary type changes require lots of additional fixes. For instance, the type `MethodDeclaration` of a node cannot be changed suddenly to `ClassDefinition` without invalidating its children (as they are *syntactically mandatory*). As many types require specific (*syntactically mandatory*) child nodes (e.g., an expression and a block for conditions), changing a type is usually impossible without invalidating the AST in general. However, for specific types and languages type changes can be reasonable and preserve the well-formedness of an AST. For example, changing an `interface` to an `abstract class` in Java can be necessary when further functionality is needed. So, when a type change occurs, it either invalidates the AST or it is performed after necessary refactorings, such as removing non-declaration statements from a method that is about to become a class definition. This ordering of changes does not need to be preserved by developers but may be established during semantic lifting. Hence, we support changing the type of nodes, when its child nodes are not invalidated, i.e., the AST is well-formed afterwards. Otherwise, this edit is perhaps better represented as a deletion followed by an insertion, i.e., replacing an existing construct. Thus, we assume the structure of the AST to remain unchanged during updates.

Renaming an AST node, i.e., changing its value, can never lead to changes in the AST as such a change is purely semantical. However, during edit script recovery renamings have to be treated with care. Consider renaming a function call `foo()` to `bar()`. We cannot know if the function `foo` was renamed or if the call to it was replaced with a call to another function `bar`. The first case would be an update edit, whereas the second should be identified as a deletion followed by an insertion. Such semantic checks would need to be made explicitly. Here existing research on variability mining [KFBA09, XXJ12, RC13, DRGP13, AGA13, KGP13, KDO14, WKP15], using type checks and even ontological heuristics to resolve exactly such dependencies among other things, can be beneficial. As the integration of such tools is out of scope of this thesis, we recap on this idea in the future work chapter in Chapter 9.

An advantage of considering updates explicitly is that existing feature mappings of edited nodes can be considered, as opposed to replacements (i.e., a deletion followed by an insertion). When a node $a \in A$ is updated, we have to ensure that this update is synchronised to all variants in which the edited node a is present. However, updating a node could only be desired for the feature described by the feature context φ. With a special feature context, a subset of all variants containing a can be identified to refine an implementation of $\mathcal{F}(a)$ for a specific feature interaction incorporating φ. Thus, we propagate the update to all variants satisfying the feature context and the previous presence condition of the node:

$$\forall a \in A.\, \mathcal{F}^*(a) \wedge \varphi \models \mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a). \tag{4.24}$$

We refer to this constraint as *legacy preservation* because it ensures the new feature mapping to incorporate the feature context and the previous feature mapping to

https://doi.org/10.24355/dbbs.084-202002271120-0

disallow arbitrary strong or unrelated feature mappings. This constraint still allows the synchronisation of the update to all variants containing a when $\varphi = \mathcal{F}(a)$ or $\varphi = \mathit{true}$. Note that *legacy preservation* extends the constraint *no guesses* for insertions with the existing feature mapping $\mathcal{F}(a)$.

Similar to insertions, the constraints *intention insurance* and *inviolacy of the living* have to be preserved as we will explain in the following. *Intention insurance* states that the feature context is satisfied in all variants that satisfy the presence condition of an edited artefact. Otherwise, variants could be identified as synchronisation targets that are incompatible to the specified feature context. Therefore, also *intention insurance* is required:

$$\forall a \in A.\, \mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) \models \varphi\,. \tag{4.25}$$

Existing nodes should not be remapped if their type or name remains unchanged, i.e., *inviolacy of the living* should be satisfied for all $a \in T_i \setminus A$ analogous to insertions. Such nodes can still be affected by feature mapping changes of their ancestors, though (e.g., when the name of a class and its feature mapping are changed, methods also inherit the changed feature mapping). Thus, when the feature mapping of an ancestor is changed, its old mapping may be overwritten, wherefore we cannot assume $\mathscr{F}_{\text{insert}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) \Rightarrow \mathcal{F}^*(a)$ as for insertions. Nevertheless, changes in presence conditions should be limited to the feature context and the existing presence condition to prevent arbitrary and unrelated mappings:

$$\begin{aligned} \models \forall a \in T_i \setminus A. \quad & \mathcal{F}(a) = \mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) \\ & \wedge \big(\mathcal{F}^*(a) \wedge \varphi \Rightarrow \mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a)\big). \end{aligned} \tag{4.26}$$

To conform the constraints, we change the feature mapping of updated nodes only. As for insertions, we do not have to assign the feature context explicitly to a node if its already satisfied by at least one of its *hierarchically mandatory* parents to avoid redundancies:

$$\begin{aligned} \mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) \coloneqq & \begin{cases} \varphi, & \varphi \neq \mathit{null}, \mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a)) \not\models \varphi, a \in A, \\ \mathcal{F}(a), & \varphi \neq \mathit{null}, \mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a)) \models \varphi, a \in A, \\ \mathcal{F}(a), & \varphi = \mathit{null}, a \in A, \\ \mathcal{F}(a), & \text{else.} \end{cases} \\ = & \begin{cases} \varphi, & \varphi \neq \mathit{null}, \mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a)) \not\models \varphi, a \in A, \\ \mathcal{F}(a), & \text{else.} \end{cases} \end{aligned} \tag{4.27}$$

If only nodes belonging to a common feature f are moved and the feature context is set to f, $\mathscr{F}_{\text{update}}$ will reasonably derive the same feature mapping as before.

Assigning just the new feature context φ instead of keeping the existing feature mapping by assigning $\mathcal{F}(a) \wedge \varphi$ to updated nodes is a design decision justified by its flexibility. As the feature context is an arbitrary formula it can also be set to $\mathcal{F}(a) \wedge \varphi$

https://doi.org/10.24355/dbbs.084-202002271120-0

to obtain the desired interaction. Furthermore, overwriting the existing feature mapping would not be possible otherwise (i.e., changing a feature mapping entirely). As assigning the feature context only is more consistent concerning the derivation upon insertions, we suppose this behaviour to be more reasonable for developers. However, opposed to insertions, we keep the existing mapping if no feature context is specified because we have no indication that the updated node should not belong to its feature anymore. Here, considering updates as an individual operation (instead of replacements) during edit script computation pays off as existing feature mappings would be lost otherwise.

In the following, we show that our feature mapping derivation upon updates $\mathscr{F}_{\text{update}}$ satisfies our imposed constraints *legacy preservation* (Constraint 4.24), *intention insurance* (Constraint 4.25), and *inviolacy of the living* (Constraint 4.26).

Theorem 4.5. *Feature mappings derived with $\mathscr{F}_{\text{update}}$ satisfy legacy preservation (Constraint 4.24) and intention insurance (Constraint 4.25).*

Proof. As both constraints apply to newly inserted nodes only, let $a \in A$ be an arbitrary but fixed updated node. By definition, there are two possible values for a derived by $\mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)$:

1. If $\varphi \neq \mathit{null} \wedge \mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a)) \not\models \varphi$, the node a will be mapped to φ. Because

$$\mathcal{F}^*(a) \wedge \varphi \models \varphi$$

is indeed a tautology, *legacy preservation* is satisfied. Furthermore,

$$\begin{aligned}\mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) &= \varphi \wedge \mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a)) \\ &\models \varphi\,.\end{aligned}$$

Thus, *intention insurance* is satisfied.

2. If $\varphi = \mathit{null} \vee \mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a)) \models \varphi$, the feature mapping of node a will remain unchanged, i.e., $\mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) = \mathcal{F}(a)$. Because

$$\mathcal{F}^*(a) \wedge \varphi \models \mathcal{F}(a)$$

is indeed a tautology, *legacy preservation* is satisfied. Furthermore,

$$\begin{aligned}\mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) &= \mathcal{F}(a) \wedge \mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a)) \\ &\models \varphi\,.\end{aligned}$$

for $\mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a)) \models \varphi$ or $\varphi = \mathit{null}$ (as defined in Section 2.3 on Page 11). Thus, *intention insurance* is satisfied.

In all possible cases (1 and 2), both constraints are satisfied. As $a \in A$ was chosen arbitrarily, both constraints are satisfied for all updated nodes $a \in A$. Thus *legacy preservation* and *intention insurance* are satisfied for feature mappings derived with $\mathscr{F}_{\text{update}}$. □

https://doi.org/10.24355/dbbs.084-202002271120-0

Theorem 4.6. *Feature mappings derived with* $\mathscr{F}_{\text{update}}$ *satisfy inviolacy of the living (Constraint 4.26).*

Proof. Let $a \in T_i \setminus A$ be an arbitrary but fixed node that was not edited. By definition,

$$\mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) = \mathcal{F}(a). \qquad \text{(referred to as (*))}$$

By assumption, the structure of tree does not change (cf., begin of this section). Thus,

$$\mathit{ancestors}_{\mathrm{T_{i-1}}}(a) = \mathit{ancestors}_{\mathrm{T_i}}(a). \qquad \text{(referred to as (**))}$$

(See Step 2iii in proof of Theorem 4.2 on Page 47 for formal definition of $\mathit{ancestors}_{\mathrm{T}}$). The new presence condition of a either is the same as before, or it changed when the feature mapping of an ancestor changed:

1. If $\mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) = \mathcal{F}^*(a)$, then
$$\mathcal{F}^*(a) \wedge \varphi \models \mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a).$$
Combined with (*), we see that *inviolacy of the living* is satisfied.

2. $\mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) \neq \mathcal{F}^*(a)$, then at least one of the propagating ancestors of a got its mapping changed, i.e., there must be a set of nodes
$$P = \{p \in \widehat{\mathit{ancestors}}_{T_i}(a) \mid \mathcal{F}(p) \neq \mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a)\} \neq \emptyset,$$
where $\widehat{\mathit{ancestors}}_T(\alpha) \coloneqq \{n \in \mathit{ancestors}_{\mathrm{T}}(\alpha) \mid \text{Propagates}(n) = \mathit{true}\}$ denotes exactly those ancestors that propagate their feature mapping. By definition of $\mathscr{F}_{\text{update}}$, we know that
$$\forall p \in P.\ \mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(p) = \varphi$$
and
$$\forall p' \in \widehat{\mathit{ancestors}}_{T_i}(a) \setminus P.\ \mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(p') = \mathcal{F}(p').$$
We can derive
$$\begin{aligned}
& \mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a) \\
= \ & \mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(a) \wedge \mathcal{F}^*(\widehat{\mathit{parent}}_{T_i}(a) \\
= \ & \mathcal{F}(a) \wedge \bigwedge_{p \in \widehat{\mathit{ancestors}}_{T_i}(a)} \mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(p) \\
= \ & \mathcal{F}(a) \wedge \Big(\bigwedge_{p \in P} \mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(p) \Big) \\
& \wedge \Big(\bigwedge_{p' \in \widehat{\mathit{ancestors}}_{T_i}(a) \setminus P} \mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)(p') \Big) \\
= \ & \mathcal{F}(a) \wedge \varphi \wedge \bigwedge_{p' \in \widehat{\mathit{ancestors}}_{T_i}(a) \setminus P} \mathcal{F}(p') \qquad \text{(because of } P \neq \emptyset\text{)} \\
= \ & \mathcal{F}(a) \wedge \varphi \wedge \bigwedge_{p' \in \widehat{\mathit{ancestors}}_{T_{i-1}}(a) \setminus P} \mathcal{F}(p'). \qquad \text{(because of (**))}
\end{aligned}$$

https://doi.org/10.24355/dbbs.084-202002271120-0

As $(\widehat{ancestors}_{T_{i-1}}(a) \setminus P) \subset \widehat{ancestors}_{T_{i-1}}(a)$, all feature mappings $\mathcal{F}(p')$ are clauses in the conjunction $\mathcal{F}^*(a)$. As also $\mathcal{F}(a)$ is part of $\mathcal{F}(a)^*$ by definition, we know that

$$\mathcal{F}(a)^* \Rightarrow \mathcal{F}(a) \wedge \bigwedge_{p' \in \widehat{ancestors}_{T_{i-1}}(a) \setminus P} \mathcal{F}(p')$$

and therefore

$$\mathcal{F}^*(a) \wedge \varphi \Rightarrow \mathcal{F}(a) \wedge \varphi \wedge \bigwedge_{p' \in \widehat{ancestors}_{T_{i-1}}(a) \setminus P} \mathcal{F}(p').$$

By substitution we obtain

$$\mathcal{F}^*(a) \Rightarrow \mathscr{F}_{\text{update}}(T_{i-1}, T_i, A, \mathcal{F}, \varphi)^*(a).$$

Combined with (*), we see that *inviolacy of the living* is satisfied.

In all possible cases (1 and 2), *inviolacy of the living* is satisfied. As $a \in T_i \setminus A$ was chosen arbitrarily, *inviolacy of the living* is satisfied for all non-updated nodes. Thus *inviolacy of the living* is satisfied for feature mappings derived with $\mathscr{F}_{\text{update}}$. □

We now can derive feature mappings upon updates of artefacts. We proved that our derivation $\mathscr{F}_{\text{update}}$ behaves as desired: It incorporates the feature context, such that variants are identified as synchronisation targets in a reasonable way, and it does not infer unrelated features to presence conditions. The feature mapping of already existing nodes remains unchanged and their presence condition is only changed if an ancestor of them got updated.

4.2 Using Feature Models for Enhancing Feature Mapping Derivation

Incorporating constraints on valid feature configurations given by a feature model introduces tighter restrictions but also opportunities. First, those constraints could render feature mappings invalid as they may break constraints. This may affect derived mappings during edits as well as the propagated feature mappings in the AST. In this section, we show that our derivations as well as propagated feature mappings in the AST always conform to the feature model. Second, feature model constraints enable simplifying existent mappings as they may unveil redundancies in mappings. Furthermore, embedding a feature model to clone-and-own development enables us to reuse product-line research.

Given a feature model, we have to ensure all derived and propagated feature mappings to conform to it if their corresponding artefacts are intended to be kept in the code base. If a mapping violates the feature model, there is no configuration satisfying it. Thus, there are no variants able to contain the mapped artefact. However, artefacts are meant to exist in certain variants, besides deliberate exceptions such as *false* upon certain deletions described in Section 4.1.5. Per assumption, existing mappings and the feature context do not violate the feature model as stated in

https://doi.org/10.24355/dbbs.084-202002271120-0

Section 4.1.2. Though, we do not assure the same for derived and propagated mappings explicitly yet. The derivations $\mathscr{F}_{\text{insert}}$, $\mathscr{F}_{\text{update}}$, and $\mathscr{F}_{\text{move}}$, are quite similar as they assign either the feature context, the old mapping, a combination of both, or *true* to artefacts. Hence, mappings derived with these three derivations satisfy any feature model as all their possible values do. The feature mapping derivation for deletions $\mathscr{F}_{\text{delete}}$, however, combines feature context φ and existing mapping $\mathcal{F}(a)$ to $\mathcal{F}(a) \wedge \neg\varphi$ if $\varphi \neq \mathit{null}$. Even negating φ can violate the feature model already, for instance if φ is a core feature. This is not an issue, though. Mappings of deleted artefacts violating the feature model just mean that the deleted artefact cannot be present in any variant anymore (just as for *false*, which is also a possible value of $\mathscr{F}_{\text{delete}}$) which does not contradict any assumptions nor intentions of us or developers. In this case, the feature mapping can be simplified to *false*. To avoid surprises, such automations should be communicated to developers, though.

Presence conditions of artefacts obtained by feature mapping propagation in the AST can never violate the feature model because the configurations of variants already conform to the constraints of a possible feature model.

Theorem 4.7. *Presence conditions obtained from feature mapping propagation inside ASTs always conform to their configuration and the global feature model.*

Proof. Each variant implements exactly one configuration of features. By assumption, these configurations are valid considering the feature model (cf., Section 3.1). Due to our *intra variant compliance* constraint, feature mappings are valid according to their configurations. Hence, feature mappings are valid considering the feature model, too. Thus, for any two feature mappings A and B valid in a certain configuration C, we know $\models \mathit{eval}(C, A)$ and $\models \mathit{eval}(C, B)$. This leads to $\models \mathit{eval}(C, A \wedge B)$. If $A \wedge B$ would violate the feature model, so would C. As C does not violate the feature model, $A \wedge B$ does neither. Hence, $A \wedge B$ is valid regarding the feature model. As the feature mapping of each node in the same AST conforms its current variant's configuration, conjuncting them also does. As conjunction is the only operation performed during AST feature mapping propagation, the AST feature mapping propagation can never violate the feature model. □

Furthermore, feature models can help in simplifying feature mappings. Generally speaking, for any partial configuration of a feature model, features that have to be de-/selected to complete the configuration can be detected, as already researched [KTS+18]. Feature mappings can be considered as such partial configurations for which mandatory (must select), forbidden (cannot select), and possible (at least / most k out of these) features can be computed. There are several ways for incorporating feature models to simplify mappings [vRGA+15] which we would like to investigate further in the future. In the following, we elaborate on two instances of such simplifications: detecting redundancies and resolving negations with alternatives in feature mappings.

Redundancies occur if mappings contain expressions that become mandatory concerning the feature model when the rest of the mapping is satisfied. For instance, feature B is redundant in mapping $A \wedge B$ when $A \Rightarrow B$ is a constraint in the feature model. Then, if A is selected, $A \wedge B$ is always true because of the modus ponens

https://doi.org/10.24355/dbbs.084-202002271120-0

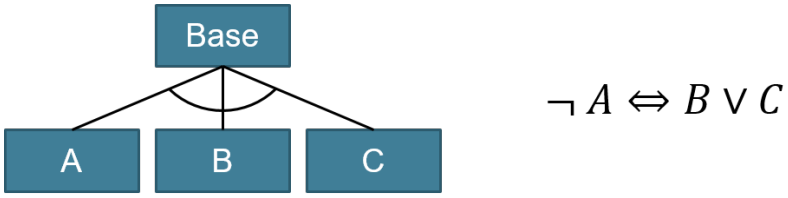


Figure 4.3: Negation Elimination Using Alternative Groups in Feature Models

axiom, i.e., $(A \wedge (A \Rightarrow B)) \Rightarrow B$. However, simplifying feature mappings automatically can result in unwanted behaviour when the feature model evolves. If the constraint $A \Rightarrow B$ is removed from the feature model, simplified feature mappings may no longer be correct and thus introduce variability bugs.

With the help of alternative constraints, negative mappings could be resolved if so wished. By excluding features from an alternative through a negative mapping, only one of the remaining features can be selected. Let $(f_1, \ldots, f_n)$ be the features of an alternative, which means exactly one of them has to be selected. Given a mapping $\mathcal{F}(a) = \neg f_i,\ i \in \{1, \ldots, n\}$ of an artefact a, then $\neg f_i \Leftrightarrow \bigvee_{j \in \{1,\ldots,n\} \setminus \{i\}} f_j$. Thus, $\mathcal{F}'(a) = \bigvee_{j \in \{1,\ldots,n\} \setminus \{i\}} f_j$ is also a valid mapping for a. This can especially be useful if $n = 2$ because then only one feature remains as a positive mapping target. However, as for redundancy detection, evolving the feature model can invalidate simplified mappings, for instance by adding a new feature f_{n+1} to the alternative group. Then $\bigvee_{j \in \{1,\ldots,n\} \setminus \{i\}} f_j$ is not equivalent to $\neg f_i$ anymore as the feature f_{n+1} is missing now. Figure 4.3 shows an example for negation elimination: Features A, B, and C are in an alternative group. Considering this feature model, the formula $\neg A$ is equivalent to $B \vee C$.

4.3 Using Other Variants for Enhancing Feature Mapping Derivation

Independent from the feature context specified by developers, the variants with the configuration they implement are already given. Thus, even when no feature context and no feature mapping are given, we still know the configuration of the currently edited variant. For instance, when deleting an unmapped artefact without a feature context (*null*) we still know that it cannot belong to the current variant anymore and thus its mapping has to evaluate to *false* in the current variant's configuration as explained in Section 4.1.5.

This enables us to derive partial knowledge on feature mappings. Partial feature mappings could serve as initial mappings or could be used to simplify mappings such as feature models do. Furthermore, depending on the number of given configurations, developers could specify feature mappings and context as a list of features instead of feature formulas such as in CIDE [KAK08] but without the necessity to explicit specify negations. In the following we give an outlook on how other variants can be used for deriving partial knowledge on feature mappings. As details on this topic are out of scope of this thesis we consider them to be a potential future work, discussed in Chapter 9.

https://doi.org/10.24355/dbbs.084-202002271120-0

Definition 4.6 (Partial Feature Mapping)**.** *We consider* $\mathcal{F}_{\text{partial}}(a)$ *to be a partial feature mapping of an artefact* $a \in \mathcal{A}$*, if and only if* $\mathcal{F}(a) \models \mathcal{F}_{\text{partial}}(a)$*, such that for all existing variants* $V \in \mathcal{V}$ *of the current project* $\models eval(C(V), \mathcal{F}_{\text{partial}}(a)) \Leftrightarrow a \in V$*.*

The presence of artefacts in variants tells us which configurations have to satisfy the feature mapping and which do not. Let $V_1, \ldots, V_n \in \mathcal{V}$ be the variants in clone-and-own development. Let $a \in \mathcal{A}$ be present in at least one variant, i.e., $a \in V_j$ for at least one $j \in \{1, \ldots, n\}$. To derive a partial mapping for $\mathcal{F}_{\text{partial}}(a)$, we differentiate between those variants $\mathcal{V}_{\text{a}}$ containing a, and those variants $\mathcal{V}_{\bar{\text{a}}}$ not containing a, such that $\mathcal{V}_{\text{a}} \dot{\cup} \mathcal{V}_{\bar{\text{a}}} = \mathcal{V}$. As variants $V \in \mathcal{V}_{\text{a}}$ contain a, their configurations $\text{C}(V)$ have to be satisfying assignments for $\mathcal{F}(a)$. Respectively, all configurations of variants in $\mathcal{V}_{\bar{\text{a}}}$ do not satisfy $\mathcal{F}(a)$. The remaining configurations $\mathcal{V}_?$ are those not associated to any variant (i.e., implicitly given by the feature model). We do not know which of these should contain the investigated artefact and which should not. Hence, we call these *concealed* assignments.

The assignments given by the variants $\mathcal{V}_{\text{a}}$, $\mathcal{V}_{\bar{\text{a}}}$, and $\mathcal{V}_?$, can be synthesised to a propositional formula γ for which we know that $\mathcal{F}(a) \Rightarrow \gamma$, thus being a partial feature mapping. The well-known Quine-McCluskey algorithm [Cur62] computes a disjunctive minimal form (DMF), the shortest form of a disjunctive normal form (DNF), according to the given assignments. Multiple solutions, i.e., feature mappings are possible. Each of those solutions is a valid partial feature mapping. As the results are DNFs, even a single clause of those is already a potential valid feature mapping. These candidates could be ranked depending on how many of the assignments $\mathcal{V}_{\text{a}}$ and $\mathcal{V}_{\bar{\text{a}}}$ evaluate as expected. For instance, for a candidate mapping $\kappa \in \mathcal{B}$ we could compute a rank by counting the number of correct assignments:

$$\text{rank}(\kappa) := \frac{\sum_{V \in \mathcal{V}_{\text{a}}} eval(\text{C}(V), \kappa) = \mathit{true} \; + \; \sum_{\bar{V} \in \mathcal{V}_{\bar{\text{a}}}} eval(\text{C}(\bar{V}), \kappa) = \mathit{false}}{|\,\mathcal{V}_{\text{a}}\,| + |\,\mathcal{V}_{\bar{\text{a}}}\,|} \tag{4.28}$$

Thus, we can obtain a partial feature mapping $\mathcal{F}_{\text{partial}}(a) = \gamma$ for any artefact $a \in \mathcal{A}$. In the future, we will investigate other research on formula recovery and synthesis. Mendonça et al. [MAaL18] even reverse engineer entire feature models by first recovering a formula, and second recovering the structure of the feature model.

Example 4.3.1. Figure 4.4 shows the feature mapping deduction from variants exemplarily. To compute a partial feature mapping, the inspected artefact `int MisterX` is first located in all variants. Second, depending on its presence in a variant, the configurations are identified as satisfying or unsatisfying assignments for $\mathcal{F}$(`int MisterX`). Their disjunction thereby is a possible feature mapping when considering the hitherto available variants only. Simplifying this disjunction yields possible feature mappings. In this case, the green feature remained as possible mapping only because it is present in exactly those variants containing `int MisterX` and the remaining features are partially included or excluded in these variants (as can be seen when comparing configurations c_4 and c_5).

Furthermore, variant knowledge could be used to simplify feature context specification. By our and other's experience, specification of an entire feature formula is error-prone similar more interlace preprocessor annotations. To address this issue,

https://doi.org/10.24355/dbbs.084-202002271120-0

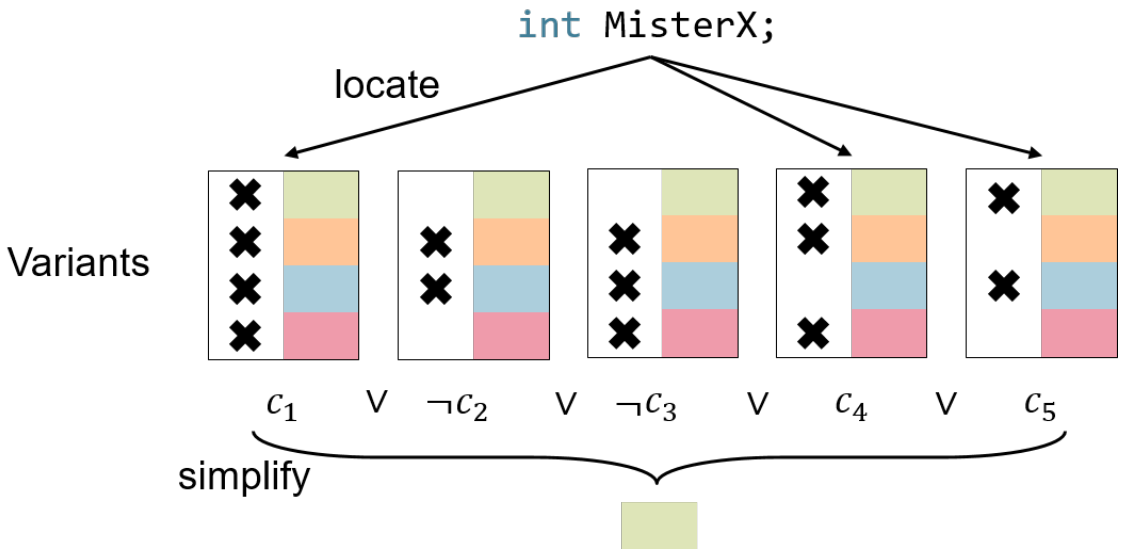


Figure 4.4: Workflow for Partial Feature Mapping Derivation from Variants

the feature context as well as the presented feature mappings could be simplified to lists of features, as done in ECCO [LELH16]. Such a list thereby contains exactly those features present in literals in the original feature formula. By restricting the partial feature mapping recovery explained in the last paragraph to such a subset of features, the derived partial mapping's accuracy can be increased because uninvolved variables can be discarded from the assignments $\mathcal{V}_{\mathrm{a}}$, $\mathcal{V}_{\bar{\mathrm{a}}}$, and $\mathcal{V}_{?}$ directly. Hence, it is irrelevant if a feature in the feature context list is part of negative or positive literals.

4.4 Known Exploits

In this section, we present two elementary exploits that in conjunction with our future variant synchronisation, enable developers to edit all variants of a clone-and-own project in a synchronised way from any single variant only. By editing a single variant, we can insert any artefact to all variants satisfying its target feature mapping. These exploits are not intended to be used by developers but show that our derivation is powerful enough to reproduce any synchronised code changes from a theoretical point of view. To fully reenact these exploits, we require a method for synchronising variants which is subject to future work and further discussed in Chapter 9.

We begin by introducing a method to queue the insertion of artefacts under any feature contexts into variants satisfying that context from any source variant in Algorithm 4.2. We refer to this procedure as *Artefact Shipping* as it broadcasts the insertion of an artefact. If the edited source variant's configuration satisfies the target feature mapping φ, we can just insert the artefact a under that feature context φ as expected. If however, the source variant's configuration does not satisfy φ, inserting a under feature context *true* and deleting it under $\neg\,\varphi$ afterwards, will yield to the artefact being mapped to $\mathit{true} \wedge \neg\neg\,\varphi = \varphi$ by $\mathscr{F}_{\mathrm{delete}}$. Depending on the future variant synchronisation, this change will be synchronised to other variants, such that a is present in exactly those variants satisfying φ.

https://doi.org/10.24355/dbbs.084-202002271120-0

Algorithm 4.2 Artefact Shipping – Procedure for Inserting Artefacts to Entire Product Line by Editing Any Single Variant Only

Given: artefact $a \in \mathcal{A}$ to insert,
arbitrary source variant $V \in \mathcal{V}$,
desired feature mapping φ for a

1: **if** $eval(\mathrm{C}(V), \varphi)$ **then**
2: insert a under feature context φ in V
3: **else**
4: insert a under feature context *null* or *true* in V
5: delete a under feature context $\neg\,\varphi$ from V
6: **end if**

Algorithm 4.3 Artefact Annihilation – Procedure for Deleting Artefacts from Entire Product Line by Editing Any Single Variant Only

Given: artefact $a \in \mathcal{A}$ to delete from all variants,
arbitrary source variant $V \in \mathcal{V}$

1: **if** $a \notin V$ **then**
2: insert a under feature context *true* in V
3: **else if** $\mathcal{F}(a) = null$ **then**
4: manually change $\mathcal{F}(a)$ to *true* ▷ e.g., be deleting and reinserting it under feature context *true*
5: **end if**

6: delete a under feature context *true* from V

To be able reproduce any code changes in a synchronised way from a single variant, we need a procedure for deleting artefacts next to insertions. Algorithm 4.3 shows how an artefact can be deleted from all clones, even from a variant that does contain that artefact. We refer to this procedure as *Artefact Annihilation* as an artefact gets removed from the entire project. To reproduce this exploit, the artefact to delete $a \in \mathcal{A}$ has to be present in our source variant and has to be mapped to *true*. To achieve this, we insert a under feature context *true* if it is not present in our variant yet. Otherwise, we have to manually set its mapping to *true* if a is not mapped yet. This could be done by deleting and reinserting it under feature context *true* for instance. Afterwards, deleting a under feature context *true* will herald its deletion from every variant as its new feature mapping is set to *false* by $\mathscr{F}_{\text{delete}}$.

Using both exploits in conjunction allows synchronising artefacts with any feature context upon variants by first deleting it with *Artefact Annihilation* and inserting them correctly with *Artefact Shipping*. While these exploits are of no direct use for developers themselves, they may prove useful for future automatism. For instance, changing the presence condition of a feature mapping manually (i.e., through a user interface and without performing edits), requires to update its presence or

https://doi.org/10.24355/dbbs.084-202002271120-0

absence in all variants depending on its new mapping. We will discuss further uses in Chapter 9.

4.5 Summary

In this chapter, we developed our method for feature mapping derivation upon edits on ASTs. As we designed our algorithm to reasonably infer mappings upon edits intuitively classified as insertion, deletion, move, or update, our derivation is independent from actual tree changes. To incorporate their domain knowledge, developer's can specify a propositional formula, called *feature context*, identifying the feature or feature interaction they are currently working on.

To ensure stepwise synchronisation between software clones, we introduced the two constraints *intra* and *inter variant compliance*. *Intra variant compliance* requires feature mappings to be valid in all variants the corresponding artefact is contained in. Thus, a variant's configuration has to be a satisfying assignment for each feature mapping present in that variant. *Inter variant compliance* requires each mapped artefact to be present in exactly those variants whose configurations are satisfying assignments for their mapping. This enforces the synchronisation of mapped software artefacts across variants. Thereby, whenever a feature mapping is introduced the corresponding artefact is queued for synchronisation.

Identifying synchronisation targets for software changes is done with according feature mappings. To ensure reasonable synchronisation, we distinguish four types of edits: insertion, deletion, move, and update. Thereby, we have shown, how domain knowledge can be preserved upon absence of the feature context and the feature context can be incorporated when it is specified.

We showed that our derivations and the AST propagation always conform to the global feature model. Further, we gave an outlook on feature models can be used to simplify existent mappings. As configurations are specified according to the feature model, propagated mappings in the AST and our derived mapping functions also do. We further presented two artificial exploits which enable inferring any feature mapping to all artefacts in a synchronised fashion from editing just a single variant. While these exploits are not intended to reflect the workflow developers should use, they are interesting from a theoretical point of view. In the future, we will investigate if they may be helpful to implement automatisms for repairing wrong feature mappings. We also have shown how knowledge of variants and their configurations can be used for enhancing feature mapping derivation. Given a mechanism to locate artefacts in variants, we are able to compute partial feature mappings from the configurations of the variants containing or not containing an artefact. Furthermore, this partial feature mapping derivation can be used to simplify feature context specification to just a list of features.

Our feature mapping derivation upon edits on ASTs enables developers to record feature mappings during their usual programming workflow by specifying the feature they are currently working on. Opposed to existing research, specifying the feature context is optional to impair development by a minimal amount only. When no feature context is specified, our derivation preserves existing mappings and is able

https://doi.org/10.24355/dbbs.084-202002271120-0

to even derive further mappings for some edits. By annotating nodes in the AST, as developed in Chapter 3, our derivation is able to consider hierarchical dependencies to further simplify derived mappings. Our algorithm is not only useful for clone-and-own development but any software development technique that requires feature mappings. Furthermore, it is independent from the actual tree diffing algorithm used for edit script computation. Optionally, semantic lifting can be employed to our algorithm to refine low-level tree edits to user-level edit scripts.

https://doi.org/10.24355/dbbs.084-202002271120-0

5. Technical Challenges

Implementing our conceptual feature mapping derivation imposes several technical challenges. In this chapter, we address those challenges or design decisions by proposing possible solutions. We identified four challenges to be of primary relevance for the implementation of our feature mapping derivation. We omit further conceptual or technical problems concerning synchronisation of variants or actual clone-and-own development, as these exceed the scope of this thesis. We present the challenges in chronological order considering their appearance in the previous Chapter 3 and Chapter 4 because there are no dependencies between them.

We start by introducing the challenge of handling redundant feature mappings in Section 5.1, which can appear due to feature mapping propagation in the AST. In Section 5.2, we propose different ways for setting up the project structure tree necessary to detect inter-file moves of artefacts and simplify feature mapping specification. We continue with our choice for visualising feature mappings in Section 5.3. Closing with Section 5.4, we show how the lift function used in Algorithm 4.1 on Page 41 can be defined to assign feature contexts to the semantic edits the user made.

5.1 Handling Redundant Feature Mappings

As introduced in Chapter 3, we map features to nodes of an AST instead of the artefact itself. Nodes inside the tree may be assigned any propositional formula as their feature mapping. When propagating feature mappings down the AST, this can lead to redundant terms in presence conditions. For instance, a method mapped to feature $A \wedge B$ inside a class mapped to A yields the redundant presence condition $A \wedge A \wedge B$ for the method. This redundancies may disturb development workflow as dealing with them can become confusing and tedious. For instance, we suppose that our feature mapping could produce unintended results if it is not made clear in the development environment (e.g., the IDE) that an artefact is mapped the same as its parent which could be invisible when both artefacts are visualised by the same colour.

For an AST node $a \in T \in \Uparrow^*$ with $a \neq \mathit{root}(T)$ and its nearest propagating ancestor $p = \widehat{\mathit{parent}}_T(a)$, we differentiate between two kinds of redundancy:

https://doi.org/10.24355/dbbs.084-202002271120-0

- $\mathcal{F}(a) \models \mathcal{F}(p)$: This is exactly the constraint whose satisfaction we enforce with AST propagation, namely that the *hierarchically mandatory* parent of a node is present when the node itself is present. If, however, this constraint is already satisfied before propagation, propagating the mappings is unnecessary and only infers redundancies to $\mathcal{F}^*(a)$. Detecting the redundant sub-expression in $\mathcal{F}(a)$ that leads to $\mathcal{F}(a) \Rightarrow \mathcal{F}(p)$ being a tautology is subject to future work and is addressed by existing research on presence condition simplification [vRGA+15].
- $\mathcal{F}(p) \models \mathcal{F}(a)$: In this case, the feature mapping $\mathcal{F}(a)$ of a is redundant because a is always present when its parent is selected, no matter what the actual mapping of a looks like. Hence, we can simplify the mapping of a to *true* (or *null*) such that it inherits the feature mapping of its parent.

Recommending possible further, more sophisticated simplifications of feature mappings in general is useful but out of scope of this thesis.

5.2 Setting up the Project Structure Tree

In Section 3.2.2, we introduced project structure trees to detect inter-file moves. Such trees describe the structure of the entire software variant and group all ASTs. An example for a project structure tree derived directly from the directory structure on disk is shown in Figure 3.4. We identified three ways to create a project structure tree for a variant in general:

1. Technical Project Structure Tree: The project structure tree could be defined with a pure technical motivation and be hidden from developers. For instance, the ASTs of all source code files could be grouped under a single central root node such that inter-file moves can be detected but no further information is held in the project structure tree. This is perhaps the most easy version of a project structure tree to implement and can be adapted to any further technical requirements.
2. Directory Structure: As in our example, the project structure can be a reflection of the hierarchical file structure of the project. Mostly, a project's semantic modules (e.g., packages in Java) are aligned according to the file structure but that does not necessarily need to be the case (e.g., namespaces in C++).
3. Build Hierarchy: To detect modules and dependencies between modules at a semantic level, build files (e.g., pom.xml for Maven, or CMakeLists.txt for CMake) could be parsed to obtain a project's structure. This is the most difficult, error-prone, and laborious strategy for constructing a project structure tree and can become arbitrarily intricate for big software projects using multiple languages or build systems. However, it bears the potential of considering further (dis-) similarities between clones for even more accurate variant synchronisation.

We consider investigating the concrete benefits of each approach to be out of scope of this thesis. To implement the derivation proposed in this thesis, creating a streamlined ghost project structure tree, is sufficient.

https://doi.org/10.24355/dbbs.084-202002271120-0

5.3 Feature Mapping Visualisation

Feature mappings can be visualised in several ways. Depending on chosen implementation approach for features (i.e., annotative or compositional), different visualisations may be adequate. For instance, in preprocessor-based or *tag-and-prune* [BCH+10, HBC+12] product lines feature mappings are directly represented by textual elements whereas in component- or plugin-based architectures mappings are visible in the project's structure.

As our feature mappings are stored externally and corresponding artefacts can be scattered arbitrarily across the software, we need to find a way for feature mapping visualisation in the editor. Existing tools handling feature mappings that way, such as CIDE [KAK08], paint the source code lines of mapped artefacts as shown in Figure 2.3. Colours are assigned to each feature and feature interactions are visualised by mixing colours.

To also visualise feature mapping propagation throughout the AST, we advocate the colourisation of entire blocks of a node as done in the Java IDE BlueJ [Lon]. BlueJ visualises a program's hierarchical structure by highlighting scopes by their type, as illustrated in Figure 5.1, to simplify programming and especially support beginners in programming. For each code element it is obviously discernible to which outer scopes it belongs. For example, the function call `toppings.add(topping)` in Line 20 is nested in a condition (blue) in a method (yellow) in a class (green). We recommend adapting this visualisation method but colour scopes not by their type but according to their feature mapping. For AST nodes that do not propagate their mapping, not the entire scope but only its begin (header and opening bracket) and end (closing bracket) could be coloured. This way, the colour sequence to the left of a line always shows the entire propagated feature mapping $\mathcal{F}^*(a)$ of the corresponding artefact a.

5.4 Lifting Feature Contexts to Edits

For our feature mapping derivation algorithm we have to map the feature context recorded by the developer to the edits in the edit script. This process is depicted by the lift function in Algorithm 4.1 on Page 41. First, we have to find a suitable way for recording the feature context during development. Second, we have to assign these contexts to the tree edits correctly.

We can achieve this by running our feature mapping derivation whenever the feature context changes. However, this requires the context to be only changeable when a valid AST can be constructed from the edited artefact. The (graphical) user interface for feature context specification would have to be disabled when the program is not in a syntactically valid state which can become confusing and tedious for developers.

Another idea to solve this problem is to record artefact edits directly in an IDE, such that edits scripts are recorded on the fly instead of recovering them retroactively with tree diffing algorithms. Though, this would require software edits to always be made in the IDE and would not be robust against external changes. Furthermore, it imposes technical challenges on how to process text changes.

https://doi.org/10.24355/dbbs.084-202002271120-0

```java
/**
 * Impeccable implementation of delicious pizza.
 */
public class Pizza
{
    private List<Topping> toppings;
    private boolean baked;

    public Pizza() {
        this.baked = false;
        this.toppings = new ArrayList<>();
    }

    public void add(Topping topping) {
        if (!baked) {
            toppings.add(topping);
        }
    }

    public void bake() {
        if (!baked) {
            for (Topping t : toppings) {
                t.bake();
            }

            baked = true;
        }
    }
}
```

Figure 5.1: Scope-Oriented Code Colorisation in BlueJ [Lon]

5.5 Summary

In this chapter, we addressed four major technical challenges for implementing our feature mapping derivation for the targeted synchronisation of variants.

First, we identified two kinds of redundant feature mappings with respect to the feature mapping propagation in ASTs. We showed how one type of redundancy can be resolved and pointed out essential other research which enables resolving the other type in a future work. Second, we proposed three methods for setting up the project structure tree of a variant and pointed out their advantages and disadvantages: a technical motivated tree, a reflection of the directory structure, and a tree reflecting the build hierarchy of the variant to reflect its semantic structure. Third, we motivated why a scope-oriented code colourisation as implemented in the Java IDE BlueJ [Lon] reasonably resolves problems in visualising feature interactions and is suitable for visualising AST-based feature mappings. Fourth, we shortly contemplated two ways for mapping user-recorded feature contexts to the edits recovered during tree diffing.

https://doi.org/10.24355/dbbs.084-202002271120-0

6. Evaluation of Applicability

In this chapter, we analyse how our feature mapping derivation supports real software development. We show which feature context has to be specified to infer desired feature mappings upon code changes. Therefore, we replay existing variability-related code change patterns extracted from a preprocessor-based software product line with a long development history. Every code changes until a certain release is classified by at least one pattern. As a baseline, we compare our derivation with the projectional product-line editor VTS by Stănciulescu et al. [SBWW16] that enables variational edits on (partial) variants related to our targeted clone-and-own scenario.

Therefore, we use the variability-related code change patterns identified by Stănciulescu et al. [SBWW16]. They analysed the history of an open-source preprocessor-based software product line and thereby extracted all code changes of code affected by variability-related preprocessor. To classify all these variability changes, Stănciulescu et al. identified a common set of change patterns. The detected patterns were cross-validated against another product line to exhaust possible variability changes.

We first formulate our research questions in Section 6.1. In Section 6.2, we describe the study design of Stănciulescu et al. for variability change pattern detection. We inspect all identified variability-related code editing patterns in Section 6.3. In Section 6.4, we answer our research questions. We discuss possible threats to validity of our analysis with respect to the set of code change patterns identified by Stănciulescu et al. [SBWW16] in Section 6.5. We summarise the chapter in Section 6.6.

6.1 Research Questions

For our evaluation, we investigate the following three research questions to check applicability of our feature mapping derivation for real software development. We do so by replaying variability-related code change patterns with our feature mapping derivation algorithm with proper feature contexts. We elaborate on our study design in detail in Section 6.2.

https://doi.org/10.24355/dbbs.084-202002271120-0

Research Question 1 — How often do developers have to switch the feature context?

With this research question we want to determine to which degree the specification of the feature context is impairing the usual programming workflow. Therefore, we investigate how often the feature context has to be switched to accomplish certain changes in variability, i.e., the feature mappings. For each variability-related code change pattern, we determine how often the feature context has to be switched and the number of different variants to edit.

Research Question 2 — How complex has the feature context to be in comparison to the desired feature mapping?

The more constraints we impose on how feature mappings can be derived, the more complex individual feature contexts might have to be. As our feature context is a propositional formula that has to be specified by the user, we want this task to be as easy as possible. Therefore, we investigate how complex the feature context has to be for the considered code change patterns in comparison to the desired mapping.

Research Question 3 — How does our feature mapping derivation compete with the projectional editor VTS by Stănciulescu et al. considering the previous two research questions?

Stănciulescu et al. identified variability-related source code editing patterns to evaluate their projectional C preprocessor software product-line variation control system VTS [SBWW16]. With VTS, (partial) variants are *checked out* by specifying a *projection*, a propositional formula over the set of features. These projected variants are then edited and *checked in* again to a central black box repository containing the entire product line. Similar to our feature context, VTS requires users to specify an *ambition* upon check-ins. As we evaluate our feature mapping derivation along the patterns identified by Stănciulescu et al., we can directly compare the complexity of our feature context with concepts of the *projection* and *ambition* in VTS.

6.2 Study Design

For our evaluation, we reuse the variability-related change patterns from the study conducted by Stănciulescu et al. for evaluation of their projectional software product-line editor VTS [SBWW16]. They analysed the history of an open-source software product line for variability-related code changes. Stănciulescu et al. described these changes with a set of common patterns they manually derived. For the rest of the chapter, we always refer to said paper when referring to the work of Stănciulescu et al. [SBWW16].

The subject system of the study by Stănciulescu et al. is the 3D printer firmware Marlin [vdZ]. Marlin is an open-source software product line written in C++ using the conditional compilation provided by the C preprocessor for variability management. Feature mappings are given by `#ifdef`, `#if`, and corresponding further macros annotating lines of code. In 2011, Marlin originated as a mixture of the existing projects *Sprinter*, also a firmware for 3D printers, and *Grbl*, a firmware for CNC

https://doi.org/10.24355/dbbs.084-202002271120-0

machines. For their study, they cloned the *MarlinDev*[1] repository from GitHub and checked out the development branch with the HEAD pointing to commit 3cfe1dce1. This version of Marlin has about 40,000 lines of code and more than 140 features in 187 source files (without the additional library files for Arduino support). Nowadays, the repository *MarlinDev* is archived but development is continued in the repository *Marlin*[2].

To identify variability-related code editing patterns, Stănciulescu et al. analysed all individual patches extracted from Marlin's commit history. They split each of the 3747 commits (without merge commits) into a patch per changed file excluding files being added, removed, or renamed. For each of the resulting 5640 patches, Stănciulescu et al. adopted a three stage process for pattern identification:

1. Randomly extract 50 commits that add or remove `#ifdef` directives by using *grep*: Each patch is inspected manually to recognise patterns.
2. Create regular expressions to represent each recognised pattern: These regular expressions are matched automatically against the pool of patches.
3. Repeat until all patches are classified: For unmatched patches, regular expressions are added and the classification is re-evaluated until each patch is classified by at least one pattern. Patches may belong to multiple patterns.

In Table 6.1, we present an extended overview by Stănciulescu et al. [SBWW16, p. 327 ff.] on the code-change patterns. The number of patches matching a given pattern, amongst possible other patterns, is given in the *#Multi* column. The *#Only* column denotes the number of patterns that exactly match only the corresponding pattern. We added the *Uniqueness* column giving the percentage of patches that could be captured by this pattern only (i.e., $100 \cdot \#Only \,/\, \#Multi$). Each pattern is examined in detail in Section 6.3.

To cross-validate the distilled patterns, Stănciulescu et al. ran their classifier on the Busybox project. They cloned the Git repository[3] at commit a83e3ae, containing about 175,000 lines of code from 13,700 commits (excluding merge commits) translatable to 34,018 patches.

Although the patterns identified by Stănciulescu et al. describe variability introduced to a preprocessor software product line, they are still valid for reasoning on our clone-and-own scenario: A possible edit in a product line always corresponds to projected edits in said product line's variants. Thus, the variability-related change patterns identified by Stănciulescu et al. correspond to edits in variants obtained from (partial) configuration of features (i.e., preprocessor definitions). In our clone-and-own scenario, we consider clones to be specific variants of a common software product as described in Section 3.1. Similarly to Stănciulescu et al., we use the patterns to reason about the corresponding projectional edit in a variant. Moreover, our feature mapping derivation is not dependent on ASTs as feature mapping target.

[1] https://github.com/MarlinFirmware/MarlinDev
[2] https://github.com/MarlinFirmware/Marlin
[3] https://git.busybox.net/busybox

https://doi.org/10.24355/dbbs.084-202002271120-0

Name	#Multi	#Only	Uniqueness in %
AddIfdef	969	129	13,31
AddIfdef^{n}	424	32	7,55
AddIfdefElse	271	4	1,48
AddIfdefWrapElse	43	17	39,53
AddIfdefWrapThen	13	3	23,08
AddNormalCode	4683	871	18,60
AddAnnotation	293	12	4,10
RemNormalCode	3932	209	5,32
RemIfdef	534	24	4,49
RemAnnotation	228	2	0,88
WrapCode	77	29	37,66
UnwrapCode	12	2	16,67
ChangePC	225	74	32,89
MoveElse	5	2	40,00

Table 6.1: Overview of Identified Variability-Related Code Editing Patterns and Their Occurrence Count in Marlin's Commit History as Determined by Stănciulescu et al. [SBWW16]

Hence, the line-based mappings given by the preprocessor annotations and distilled to edit patterns are no obstacle.

Stănciulescu et al. identified these patterns to validate their own projectional variation control system VTS [SBWW16]. VTS allows editing preprocessor-based software product lines such that developers can work on partially preprocessed variants instead of the entire product line at once. Figure 6.1 depicts the differences and commonalities between our feature mapping derivation for clone-and-own development and VTS. Whereas we want to enhance clone-and-own with concepts from product-line engineering, VTS projects the product line to a (partial) clone to simplify product-line editing. They refer to it as *projectional editing*[4]. After editing, the projected variant has to be reintegrated into the product line. This step is quite similar to our derivation because feature mappings have to be derived from code changes and a user-defined *ambition* or feature context, respectively.

Figure 6.2 gives a detailed overview of the workflow in VTS. The product line is stored in a central black box repository r. With the *get* function developers can retrieve a view v of the product line. Therefore, developers have to specify a *projection* p, a propositional formula over the set of features describing the partial configuration of features they want to edit. VTS generates the view v of r by iterating over all boolean preprocessor annotations. It omits annotated code if its presence condition contradicts the projection (also considering `#if` and `#else` blocks individually). If neither the `#if` block nor its alternative `#else` block contradicts the projection,

[4]Originally, *projectional editing* refers to dedicated and specialised editors that present users a view or projection of the actual artefact to edit (e.g., modifying the AST in a graphical editor instead of the textual representation of source code). Contrary, Stănciulescu et al. classify their partial pre-compilation as *projectional editing* which is not to be confused with its original meaning.

https://doi.org/10.24355/dbbs.084-202002271120-0

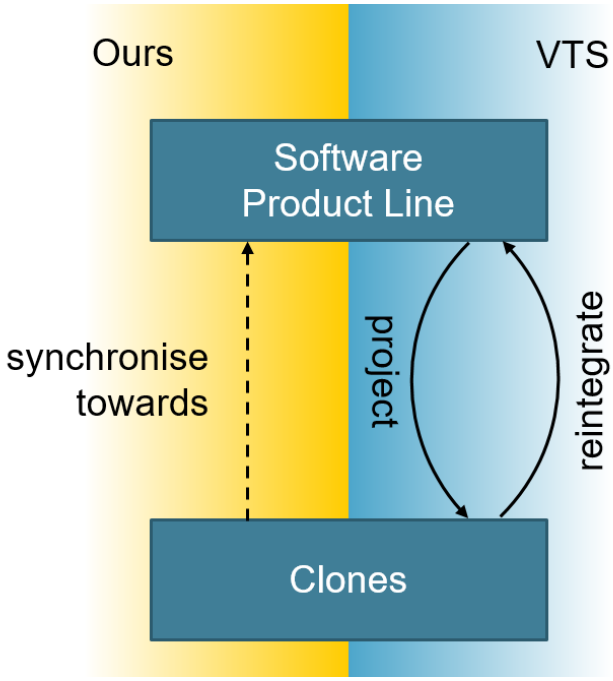


Figure 6.1: Goal Comparison of Our Feature Mapping Derivation for Clone-and-Own Development With VTS

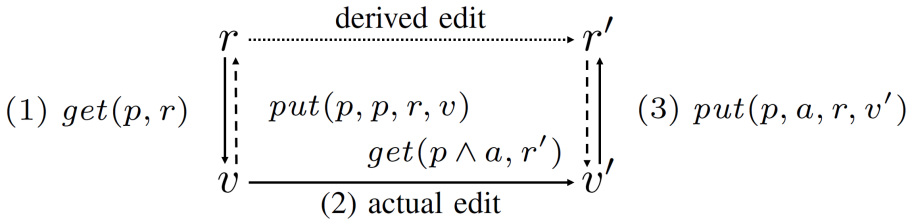


Figure 6.2: Projectional Product-Line Editing Workflow in VTS [SBWW16, p. 325]

both remain in the view. Developers can submit their edited version v' of the view v back to the repository r with the *put* function, leading to a new repository revision r'. To reintegrate changes, an *ambition* a has to be specified for the *put* function. An ambition is a propositional formula over the set of features describing which feature or feature interaction was edited, similar to our feature context. Because no edit in a view projected by p can affect code outside of that view, the entire edit is mapped to $p \wedge a$ internally. To obtain a more compact representation of the edit and avoid redundancy, Stănciulescu et al. apply several minimisation rules to the inferred preprocessor directive $p \wedge a$. The dashed lines in Figure 6.2 indicate the idempotency of the *put* and *get* function following each other: Applying a *get* immediately after a *put* operation or a *put* immediately after a *get* operation does not change the original repository and the view respectively.

6.3 Variability-Related Code Editing Patterns

In this section, we examine all variability-related code editing patterns identified by Stănciulescu et al. [SBWW16]. These patterns are the kinds of edits, our feature mapping derivation needs to support. For clarity, we adopt their classification of

https://doi.org/10.24355/dbbs.084-202002271120-0

the 14 patterns into the three categories: *Code-Adding Patterns*, *Code-Removing Patterns*, and *Annotation-Change Patterns*[5]. These categories apply to software product-line development and thereby must not necessarily correspond to the same edit operations in variant (i.e., clone-and-own) development. We show how each pattern can be described by edit operations in one or more variants supervised by our derivation. If multiple possibilities for reproducing a pattern in our clone-and-own scenario exist, we present the simplest one we could identify in terms of number of edits and divergence of the feature context from the desired feature mapping. Furthermore, we compare our derivation to the projectional variation control system VTS by Stănciulescu et al. [SBWW16].

We also present a projectional workflow in VTS for each pattern as a baseline for our derivation. As a fallback, it is always (i.e., for each pattern) possible in VTS to use the projection *true* and edit the whole product line at once. However, this is not the goal of VTS and is also not applicable to our clone-and-own scenario.

To visualise each pattern we use the *unified diff* notation. Added lines are labeled with a plus (+). Removed lines are labeled with a minus (−). Lines without marker remain unchanged. Just as Stănciulescu et al., we omit further meta information as it is neither relevant nor useful for pattern representation.

We begin with code-adding patterns in Section 6.3.1 and continue with the code-removing patterns in Section 6.3.2. The remaining annotation-change patterns are examined in Section 6.3.3.

6.3.1 Code-Adding Patterns

We first cover those patterns related to insertions of code with or without adherent feature mappings into a software product line. In Table 6.2, we show an overview of code insertion patterns. The column *projection* denotes the projections necessary for reproducing the corresponding pattern in VTS by Stănciulescu et al. [SBWW16]. Similarly, the *ambition* column denotes the necessary ambition for the edit in VTS. Analogous to projection and ambition, the column *feature context* contains the feature contexts that have to specified in our derivation to reproduce the pattern. The literal U is a shorthand for a frequently used feature in the following patterns. Pattern *AddIfdef*^{n} requires more sophisticated conditions $c_i,\ i \in \{1,\ldots,n\}$ and $\varphi_i,\ i \in \{1,\ldots,j\}$ that are explained in detail in the corresponding section. The number $j \in \mathbb{N}$, with $j \leq n$, denotes the number of individual projections, ambitions, and feature contexts that are necessary to reproduce the pattern *AddIfdef*^{n}. Thereby, *true*^{j} for the projection indicates that the projection has to be *true* for all j commits. The *#commits* indicates how many complete edit cycles consisting of retrieving a view with *get*, editing it, and submitting it with *put* back to the repository are necessary in VTS to reproduce a pattern. If multiple commits are necessary, all corresponding projections and ambitions are given in the corresponding order. In column *#variants to edit* we indicate how many variants have to be edited in our

[5]In the original paper, *Annotation-Change Patterns* were referred to as *Other Patterns*. As all these other patterns are related to changes in presence conditions and not code, we use a more precise name.

https://doi.org/10.24355/dbbs.084-202002271120-0

Pattern Name	Target feature mapping	VTS: projections	VTS: ambitions	VTS: #commits	our derivation: feature contexts	our derivation: #variants to edit	our derivation: involved derivations
AddIfdef	U	*true*	U	1	U	1	$\mathscr{F}_{\text{insert}}$
AddIfdef$^{\text{n}}$	$c_1, \ldots, c_n$	$true^j$	$\varphi_1, \ldots, \varphi_j$	j	$\varphi_1, \ldots, \varphi_j$	$\leq j$	$n \times \mathscr{F}_{\text{insert}}$
AddIfdefElse	$U, \neg U$	*true*, $\neg U$	$U, \neg U$	2	$U, \neg U$	2	$2 \times \mathscr{F}_{\text{insert}}$
AddIfdefWrapElse	$U, \neg U$	*true*	U	1	U	1	$\mathscr{F}_{\text{delete}}, \mathscr{F}_{\text{insert}}$
AddIfdefWrapThen	$U, \neg U$	*true*	$\neg U$	1	$\neg U$	1	$\mathscr{F}_{\text{delete}}, \mathscr{F}_{\text{insert}}$
AddNormalCode	*true* or U	*true* or U	*true*	1	*true* or U	1	$\mathscr{F}_{\text{insert}}$
AddAnnotation	–	— not applicable —					

Table 6.2: Variability-Related Code-Adding Patterns

clone-and-own scenario. In the last column, *necessary derivations*, we list the individual derivation functions involved when applying our derivation algorithm given in Algorithm 4.1.

Pattern 1 – *AddIfdef*

The first editing pattern covers the insertion of code with a surrounding preprocessor annotation:

```
+ #ifdef ULTRA_LCD
+   lcd_setalertstatuspgm(lcd_msg);
+ #endif
```

Figure 6.3: Pattern *AddIfdef* (Adapted From [SBWW16, p. 327])

The code is added with a feature mapping. In this example, the inserted code line is mapped to `ULTRA_LCD`. In the following, we use the abbreviations *lcd* for the code in the second line and U for the feature `ULTRA_LCD`. This pattern can be reproduced in any clone whose configuration is a satisfying assignment for the `#ifdef` condition φ (e.g., $\varphi = U$). In such a clone, inserting code under feature context φ would reproduce this pattern. Our algorithm derives the mapping with $\mathscr{F}_{\text{insert}}$ and thereby assigns φ to the inserted code fragment as described by this pattern.

Stănciulescu et al. suggest to check out the partial variant given by the trivial projection *true* (i.e., the whole product line) in VTS. The line of code *lcd* is added and checked in under the ambition U:

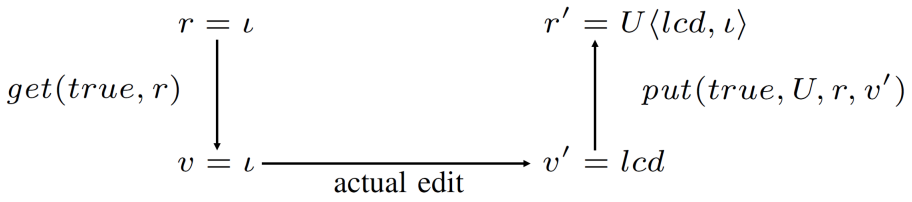


Figure 6.4: Workflow for Pattern AddIfdef in VTS [SBWW16, p. 327]

https://doi.org/10.24355/dbbs.084-202002271120-0

To simplify the visualisation of the workflow in VTS, Stănciulescu et al. assume starting with an empty repository $r = \iota$ for reproducing *AddIfdef* and all following patterns as shown above in Figure 6.4. The notation $U\langle lcd, \iota\rangle$ describes code *lcd* being active when feature U is chosen and the previous repository content ι (i.e., nothing) to be present elsewise (i.e., on $\neg U$).

Pattern 2 – *AddIfdef*^{n}

This pattern[6] groups multiple applications of the previous *AddIfdef* with conditions $c_1, \ldots, c_n,\ n \geq 2$. As before, such an edit is possible with successive insertions of source code artefacts under corresponding feature contexts. As not all conditions have to be pairwise disjunct (i.e., $c_i \neq cj,\ i, j \in \{i, \ldots, n\},\ i < j$), the feature context has to be changed at most n times. Thus, we have to repeat pattern *AddIfdef* j times with contexts $\varphi_1, \ldots, \varphi_j$ and $j \leq n$, such that for each unique condition $c_k,\ k \in \{1, \ldots, n\}$ there is exactly one feature context $\varphi_{k'} = c_k,\ k' \in \{1, \ldots, j\}$ and each feature context corresponds to one of the initial conditions. In the worst case, all j feature contexts pairwise contradict each other, or are exclusive to a unique existing clone each (e.g., when there are not enough different configurations implemented as clones). Then, a different variant for each feature context would have to be edited. In the best case, all j feature contexts are valid for a single implemented variant in which all edits can be made at once.

Similarly, this pattern can be reproduced in VTS by multiple applications of the *AddIfdef* pattern. For each of the unique conditions $\varphi_1, \ldots, \varphi_j$ one checkout/checkin sequence is necessary. Alternatively, all annotations could be integrated manually into the entire product line at once with projection and ambition set to *true*.

Pattern 3 – *AddIfdefElse*

Similar to *AddIfdef*, code surrounded by an `#ifdef` is added but extended by a following `#else` statement:

```
+ #ifdef ULTRA_LCD
+   lcd_setalertstatuspgm(lcd_msg);
+ #else
+   alertstatuspgm(msg);
+ #endif
```

Figure 6.5: Pattern *AddIfdefElse* (Adapted From [SBWW16, p. 328])

In the following, we abbreviate the code fragment `alertstatuspgm(msg);` with *alert*. The two inserted code fragments have the feature mappings $\mathcal{F}(lcd) = U$ and $\mathcal{F}(alert) = \neg U$. We cannot reproduce this pattern directly in a single variant because both feature mappings are alternative (i.e., mutual exclusive). Each line has to be added in a variant whose configuration satisfies the respective feature

[6] In the original paper, this pattern was referred to as AddIfDef*. We quantify the count of applications of the AddIfdef pattern by n to be able to reason on this pattern more accurately.

https://doi.org/10.24355/dbbs.084-202002271120-0

mapping. The same as for *AddIfdef*, each code fragment has to be inserted with a corresponding feature context such that $\mathscr{F}_{\text{insert}}$ will be used to assign the feature context as mapping. Notably, the second code fragment *alert* has to be inserted under feature context $\neg U$. We do not consider the lack of direct replicability of this pattern to be a disadvantage. Both lines of code are never supposed to appear simultaneously in a single variant and clones are intended to be variants instead of a whole software product line.

In VTS, this pattern can most simply be reproduced by checking out a view with projection *true*, adding the entire block (with preprocessor annotations) directly, and checking it in with the ambition *true*. A projectional workflow requires one edit per branch (i.e., two). First, *lcd* is added with ambition U. In a second commit, *alert* is added with ambition $\neg U$.

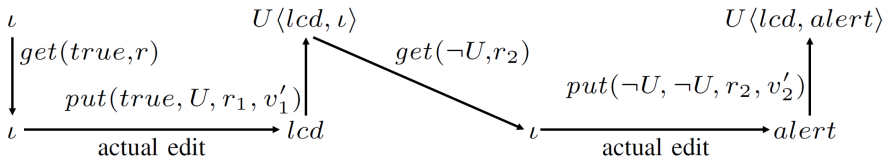


Figure 6.6: Workflow for Pattern AddIfdefElse in VTS [SBWW16, p. 328]

Pattern 4 – *AddIfdefWrapElse*

The following pattern introduces a feature mapping to a formerly unmapped artefact by surrounding it with an `#else` branch of a new preprocessor condition:

```
+ #ifdef ULTRA_LCD
+   lcd_setalertstatuspgm(lcd_msg);
+ #else
    alertstatuspgm(msg);
+ #endif
```

Figure 6.7: Pattern *AddIfdefWrapElse* (Adapted From [SBWW16, p. 328])

To reproduce this pattern we have to edit a variant V that implements feature U (i.e., $eval(\mathrm{C}(V), U) = true$) and is not supposed to contain *alert* anymore. By deleting *alert* under feature context U it will be mapped to $\neg U$ by $\mathscr{F}_{\text{delete}}$. Afterwards, the new code *lcd* can be inserted under the same feature context U to map it to the same with $\mathscr{F}_{\text{insert}}$.

In VTS, *alert* can be removed from the product line (i.e., with projection *true*) and replaced with *lcd*. Submitting it to the repository under ambition U will mark the replacement only valid for variants containing feature U:

https://doi.org/10.24355/dbbs.084-202002271120-0

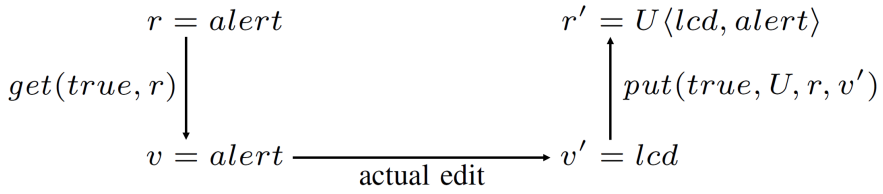


Figure 6.8: Workflow for Pattern AddIfdefWrapElse in VTS [SBWW16, p. 328]

Pattern 5 – *AddIfdefWrapThen*

The reciprocal case of *AddIfdefWrapElse* is the *AddIfdefWrapThen* pattern in which a code fragment gets annotated while also adding an `#else` case:

```
+ #ifdef ULTRA_LCD
    lcd_setalertstatuspgm(lcd_msg);
+ #else
+   alertstatuspgm(msg);
+ #endif
```

Similarly, this pattern can be reproduced the same way as *AddIfdefWrapElse* but with the inverse feature context $\neg U$ (instead of U). In a variant containing the code *lcd* and satisfying $\neg U$, deleting *lcd* under feature context $\neg U$ will let $\mathscr{F}_{\text{delete}}$ determine $\neg\neg U = U$ as the new mapping. Subsequently, *alert* has to be inserted under the same feature context $\neg U$.

The workflow for VTS stays the same as for pattern *AddIfdefWrapElse* but with the ambition negated to $\neg U$.

Pattern 6 – *AddNormalCode*

This patterns comprises the insertion of code without any associated feature mapping (i.e., under no feature context). The inserted code is either non-variational (1):

```
+ lcd_setalertstatuspgm(lcd_msg);
```

or is supposed to exist under certain already formulated presence condition (2):

```
  #ifdef ULTRA_LCD
+   lcd_setalertstatuspgm(lcd_msg);
  #endif
```

Non-variational code is supposed to be contained in every variant. To reproduce case (1), the code fragment can be inserted with feature context *true* so that its insertion will be synchronised to every other variant.

To reproduce case (2), the new code has to be inserted under feature context U just as for the *AddIfdef* pattern. When an outer scope is already mapped to U (e.g., a class or method), the new code can even be inserted without any feature context

https://doi.org/10.24355/dbbs.084-202002271120-0

Pattern Name	Target feature mapping	VTS: projections	VTS: ambitions	VTS: #commits	our derivation: feature contexts	our derivation: #variants to edit	our derivation: involved derivations
RemNormalCode	*false*	U	U	1	*true* (or *null*)	1	$\mathscr{F}_{\text{delete}}$
RemIfdef	*false*, *false*	$U, \neg U$	$U, \neg U$	≤ 2	*true* or *null*	≤ 2	$(2\times)\,\mathscr{F}_{\text{delete}}$
RemAnnotation	–	— not applicable —					

Table 6.3: Variability-Related Code-Removing Patterns

(i.e., *null* our *true*). It will inherit the outer scope mapping due to feature mapping propagation in the AST.

In VTS, reproducing case (1) (i.e., non-variational code) is done with projection and ambition both being *true* in a single check-in check-out cycle because the non-variational code has to be present in every variant. For case (2), a view with the projection being the presence condition (e.g., U in this case) has to be checked out. The code *lcd* can then be inserted and and checked in with the ambition *true*.

Pattern 7 – *AddAnnotation*

In this pattern, whitespace changes (e.g., adding a comment) occur in annotations or single preprocessor statements are added to the code. This usually happens when syntactically ill-formed annotations are repaired (e.g., adding a missing `#endif`). Depending on the chosen representation of feature mappings, ill-formed feature mappings may be warded in the first place (e.g., when using colours). As we allow synchronisation between variants for well-formed mappings only, we do not consider this case further.

Similarly, this pattern is not applicable in VTS, too. In VTS, only views containing well-formed preprocessor annotations can be checked in.

6.3.2 Code-Removing Patterns

In this section, we cover code editing patterns in which code with or without adherent feature mappings gets removed from a software product line. We give an overview of these patterns in Table 6.3 similar to the code-adding patterns.

Pattern 8 – *RemNormalCode*

This pattern depicts the removal of a code fragment, regardless of whether it is annotated (i.e., mapped to a feature) or not:

```
#ifdef ULTRA_LCD
-   lcd_setalertstatuspgm(lcd_msg);
    alertstatuspgm(msg);
#endif
```

Figure 6.9: Pattern *RemNormalCode* (Adapted From [SBWW16, p. 328])

https://doi.org/10.24355/dbbs.084-202002271120-0

As artefacts are removed from the entire software product line this way, they also have to disappear from each variant containing them. Deleting *lcd* under feature context *true* (or even *null* if $\mathcal{F}(lcd) \neq null$) will yield its removal in every variant because of

$$\begin{aligned}\mathscr{F}_{\text{delete}}(\dots)(lcd) &= \mathcal{F}(lcd) \wedge \neg\, true \\ &= \mathcal{F}(lcd) \wedge false \\ &= false\,.\end{aligned}$$

In VTS, this pattern can be reproduced by retrieving the projection U, deleting code *lcd*, and check it in under the same ambition U:

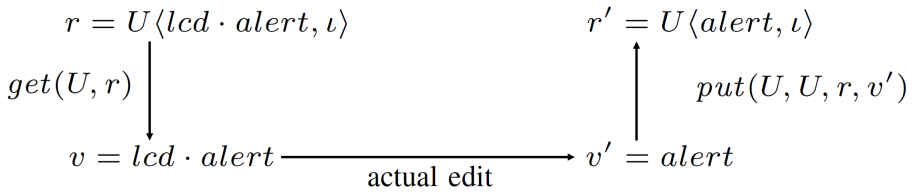


Figure 6.10: Workflow for Pattern RemNormalCode in VTS [SBWW16, p. 328]

The dot · denotes the concatenation of two code fragments (i.e., both code fragments follow each other in the document).

Pattern 9 – *RemIfdef*

This pattern comprises the removal of entire preprocessor conditions. It covers annotations with and without adherent `#else` branch.

```
- #ifdef ULTRA_LCD
-   lcd_setalertstatuspgm(lcd_msg);
- #else
-   alertstatuspgm(msg);
- #endif
```

Figure 6.11: Pattern *RemIfdef* (Adapted From [SBWW16, p. 329])

We can reproduce this pattern by deleting the source code fragments from variants containing them. As both cases are mutually exclusive, we have to edit a different variant for each branch, as for *AddIfdefElse* in Section 6.3.1. In this pattern, the artefacts get removed from the entire software product line and thereby should no longer exist in any variant, such as for the previous *RemNormalCode* pattern. As the artefacts already have a feature mapping, deleting them without any feature context (i.e., *null*) is sufficient because $\mathscr{F}_{\text{delete}}$ will evaluate to *false* and thereby herald their deletion from every variant. As well, the feature context can be set to the feature mapping itself to achieve the same result.

https://doi.org/10.24355/dbbs.084-202002271120-0

Pattern Name	Target feature mapping	VTS: projections	VTS: ambitions	VTS: #commits	our derivation: feature contexts	our derivation: #variants to edit	our derivation: involved derivations
WrapCode	U	$true$	$\neg U$	1	$\neg U$	1	$\mathscr{F}_{\text{delete}}$
UnwrapCode	$true$	$\neg U$	$\neg U$	1	$true$	1	$\mathscr{F}_{\text{insert}}$
ChangePC	φ_2	$\neg\varphi_1, true$	$\neg\varphi_1, \neg\varphi_2$	2	$null, \varphi_2$	≤ 2	$\mathscr{F}_{\text{delete}}, \mathscr{F}_{\text{insert}}$
MoveElse	U	$\neg U, U$	$\neg U, U$	2	$null, U$	2	$\mathscr{F}_{\text{delete}}, \mathscr{F}_{\text{insert}}$

Table 6.4: Variability-Related Annotation-Change Patterns

For VTS, this pattern is dual to *AddIfdefElse* in Section 6.3.1. In that sense, it can be reproduced with two consecutive edits, each corresponding to insertion of the code belonging to one of the branches:

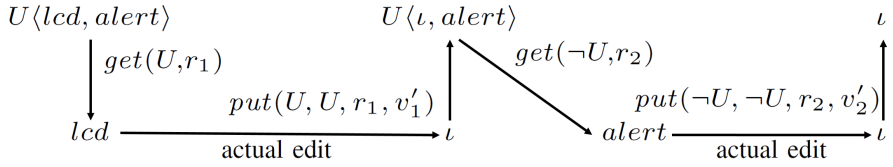


Figure 6.12: Workflow for Pattern RemIfdef in VTS [SBWW16, p. 329]

Pattern 10 – *RemAnnotation*

Dual to pattern *AddAnnotation* in Section 6.3.1, this pattern occurs when single annotations are removed. This either repairs or yields to syntactically ill-formed preprocessor annotations. As before, we do not consider ill-formed annotations as we do not synchronise them and avoid them with a proper representation right away. The same is true for VTS, which does not support ill-formed annotations either.

6.3.3 Annotation-Change Patterns

We complete the pattern examination with the remaining four patterns that are not associated to code changes but annotation changes. In Table 6.4, we give an overview of the patterns as in the previous sections.

Pattern 11 – *WrapCode*

In this pattern, previously unmapped code gets annotated by surrounding it with an `#ifdef` and closing `#endif` statement:

```
+ #ifdef ULTRA_LCD
    lcd_setalertstatuspgm(lcd_msg);
+ #endif
```

Figure 6.13: Pattern *WrapCode* (Adapted From [SBWW16, p. 329])

To reproduce this pattern, we have to remove the code fragment from those variants that should not contain it anymore. Therefore, we have to delete `lcd` under feature

https://doi.org/10.24355/dbbs.084-202002271120-0

context $\varphi = \neg U$ from a variant V containing it, i.e., with $eval(\mathrm{C}(V), \varphi) = true$. Then, $\mathscr{F}_{\text{delete}}$ will assign $\mathcal{F}(\texttt{lcd}) \wedge \neg\neg\varphi = \varphi = U$ to `lcd` because its previous mapping $\mathcal{F}(\texttt{lcd})$ is either *null* or *true*. Deleting `lcd` under context $\varphi = \neg U$ is reasonable because it indicates that `lcd` does not belong to $\neg U$ anymore and thus to U.

As this pattern changes the feature mapping of an artefact from *true* or *null* to a more concrete mapping, we consider this pattern to be a special case of the pattern *ChangePC*, introduced later in this section. This pattern also covers the case of the code fragment being mapped to *true* beforehand (instead of *null*).

This pattern can be reproduced similarly in VTS as shown below in Figure 6.14. The code *lcd* that should be wrapped has to be removed from all variants not satisfying its new feature mapping. Therefore, the entire product line is checked out with the trivial projection *true* and the code removed. Submitting the results under ambition $\neg U$ indicates the changes being exclusive to those variants satisfying $\neg U$ only. Other variants remain thus unchanged.

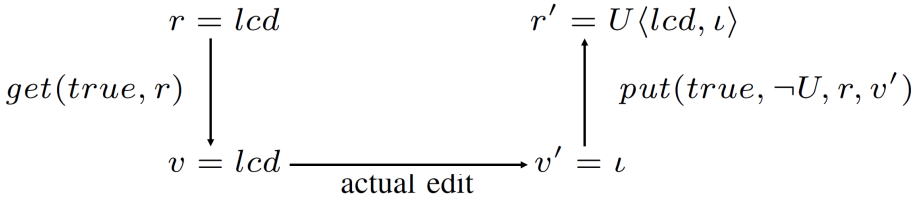


Figure 6.14: Workflow for Pattern WrapCode in VTS [SBWW16, p. 329]

Pattern 12 – *UnwrapCode*

Dual to the previous pattern *WrapCode*, preprocessor annotations surrounding artefacts are removed in this pattern:

```
- #ifdef ULTRA_LCD
    lcd_setalertstatuspgm(lcd_msg);
- #endif
```

Figure 6.15: Pattern *UnwrapCode* (Adapted From [SBWW16, p. 329])

The new feature mapping of the artefact is *true* (but its actual presence condition might depend on further outer scope annotations), meaning that it has to be present in every variant. Therefore, we have to insert `lcd` into every variant that satisfies its new presence condition and does not contain it yet. We can achieve this by inserting `lcd` under feature context $\varphi = true$ such that `lcd` will be mapped to *true* by $\mathscr{F}_{\text{insert}}$.

With convenient tool support, we could also allow feature context $\varphi = \neg U$. Upon synchronisation, the new feature mapping U (derived with $\mathscr{F}_{\text{insert}}$) has to be merged with the current mapping $\mathcal{F}(\texttt{lcd}) = U$ to the desired feature mapping *true*.

We deliberately disallow removing feature mappings entirely (i.e., setting them to *null*) because we aim to improve consistency and synchronisation between clones

https://doi.org/10.24355/dbbs.084-202002271120-0

trough a growing amount of mapped artefacts. Instead, whenever a feature mapping has to be removed because it is not (or no longer) correct, it can be changed instead. Therefore, we consider this pattern to be a special case of the following pattern *ChangePC*.

Stănciulescu et al. admit that *"this pattern is not very amenable to [their] projectional editing model"* in VTS [SBWW16, p. 329]. As for our clone-and-own scenario, the code to unwrap has to be inserted to each variant not containing it yet. Therefore, the view with the projection being the negated feature mapping $\neg U$ has to be checked out. The code *lcd* has to be inserted and checked in with the ambition being the same as the projection. The resulting choice in the product line $U\langle lcd, lcd\rangle$ (i.e., choose *lcd* when U is satisfied or *lcd* when it is not) is simplified by the implemented minimisation rules to just *lcd* in the new revision r':

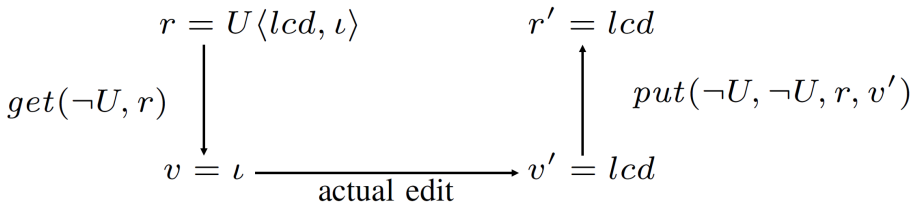


Figure 6.16: Workflow for Pattern UnwrapCode in VTS [SBWW16, p. 329]

As projection and ambition are reasonable but not very intuitive, Stănciulescu et al. suggest a non-projectional workflow here [SBWW16, p. 329]. When obtaining the entire product line with projection *true*, the preprocessor annotations can be removed manually and checked in again with the ambition *true*. This also avoids the need to exactly duplicate the code *lcd* but forfeits the benefits of projectional editing.

Pattern 13 – *ChangePC*

This patterns captures cases where the presence condition of a preprocessor annotation is changed:

```
- #ifdef ULTRA_LCD
+ #if ULTRALCD && ULTIPANEL
    lcd_setalertstatuspgm(lcd_msg);
  #endif
```

Figure 6.17: Pattern *ChangePC* (Adapted From [SBWW16, p. 329])

Here, not only the feature `ULTIPANEL` was added to the condition but also `ULTRA_LCD` was renamed to `ULTRALCD`. Reproducing this pattern with our derivation mechanism depends on our synchronisation mechanism (cf., Chapter 9). We refer to the previous annotation `ULTRA_LCD` as φ_1 and to the new annotation `ULTRALCD && ULTIPANEL` as φ_2. Currently, we have to delete the affected code entirely (i.e., removing it under feature context $\varphi_{del} = null$ or $\varphi_{del} = \varphi_1$ such that $\mathscr{F}_{\text{delete}}$

https://doi.org/10.24355/dbbs.084-202002271120-0

will assign *false* as its feature mapping) and reinsert it under the new condition $\varphi_{ins} = \varphi_2 =$ `ULTRALCD && ULTIPANEL`.

Changing the feature mapping $\mathcal{F}(a)$ of an artefact $a \in \mathcal{A}$ to $\mathcal{F}'(a)$ requires the following subsequent variant synchronisation:

- In variants V satisfying the old mapping $\mathcal{F}(a)$ but not the new one $\mathcal{F}'(a)$ (i.e., $eval(\mathrm{C}(V), \mathcal{F}(a)) = true$ and $eval(\mathrm{C}(V), \mathcal{F}'(a)) = false$), the artefact a has to be removed.
- In variants V satisfying the new mapping $\mathcal{F}'(a)$ but not the old one $\mathcal{F}(a)$ (i.e., $eval(\mathrm{C}(V), \mathcal{F}(a)) = false$ and $eval(\mathrm{C}(V), \mathcal{F}'(a)) = true$), the artefact a has to be inserted.

In VTS, this pattern can be reproduced similarly. By applying both previous patterns, *UnwrapCode* followed by *WrapCode*, the old annotation (e.g., U) can be removed and replaced in a second commit cycle with the new presence condition (e.g., `ULTRALCD && ULTIPANEL`). This workflow is also examined in the similar pattern *MoveElse*, illustrated in the next subsection. Stănciulescu et al. admit that this pattern is *"perhaps better supported without a projectional edit"* [SBWW16, p. 329] by checking out the entire product line with projection *true* and changing the preprocessor annotation manually. The edited variant can then be checked in again with ambition *true*.

As we store feature mappings offline (i.e., not explicitly in the text but separately), renaming features can be done centrally (e.g., in the global feature model). However, the features have to be renamed in all files containing meta information of feature mappings. To avoid this error-prone task, we suggest identifying features by unique indexes instead by their name. And index-to-name mapping can be stored in a central global register together with the feature model.

Pattern 14 – *MoveElse*

In this pattern, code is moved from one branch of a condition to another one by moving an `#else` statement.

```
  #ifdef ULTRA_LCD
    lcd_setalertstatuspgm(lcd_msg);
- #else
    alertstatuspgm(msg);
+ #else
    cleanup(msg);
  #endif
```

Figure 6.18: Pattern *MoveElse* (Adapted From [SBWW16, p. 329])

This (rare) pattern is not very amenable to our derivation. Because old and new mapping of the affected artefact *alert* are mutual exclusive, we have to edit two

https://doi.org/10.24355/dbbs.084-202002271120-0

variants (i.e., one satisfying the old and one satisfying the new feature mapping). First, we have to remove *alert* entirely such that it is not present in any variant anymore. Similar to pattern *ChangePC*, this can be done by deleting *alert* under feature context $\varphi_{del} = true$ such that $\mathscr{F}_{\text{delete}}$ maps it to $false$. Second, we have to insert *alert* again under the feature context U such that it is mapped to U by $\mathscr{F}_{\text{insert}}$. Otherwise, this pattern can also be reproduced inversely by first inserting it to each variant according to pattern *UnwrapCode*, and second deleting it from all original variants (i.e., those satisfying $\neg U$) under feature context $\varphi'_{del} = \neg U$. It is then mapped to $\mathcal{F}(alert) \wedge \neg\, \varphi'_{del} = true \wedge \neg\neg U = U$ by $\mathscr{F}_{\text{delete}}$.

In VTS, this pattern can be reproduced the same as the previous pattern *ChangePC*. First, *alert* has to be removed entirely (i.e., from the partial variant projected by $\neg U$). Afterwards, *alert* has to be reinserted under the opposite presence condition U. In the following workflow overview, the three code lines from the above example are abbreviated by their initial letters l, a, and c, respectively:

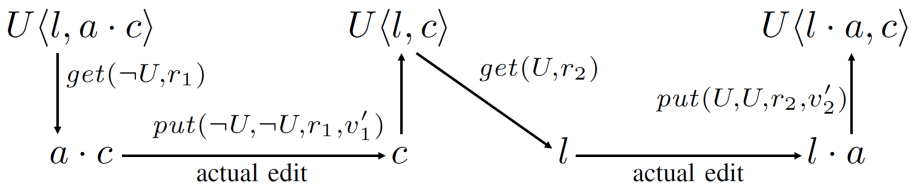


Figure 6.19: Workflow for Pattern MoveElse in VTS [SBWW16, p. 330]

6.4 Discussion

In this section, we answer our research questions formulated in Section 6.1 based on our study conducted in the previous section. We discuss each question separately. Therefore, we refer to our results summarised in Table 6.2, Table 6.3, and Table 6.4 throughout this section. Thereby, we do not consider the patterns *AddAnnotation* and *RemAnnotation* because both are neither applicable with our derivation nor with VTS by Stănciulescu et al. [SBWW16]. Both patterns capture whitespace changes, and introducing or repairing ill-formed preprocessor annotations neither of which are recognised, supported, or necessary for both our derivation and VTS.

6.4.1 RQ 1 – Count of Feature Context Switches

Our first research question is: How often do developers have to switch the feature context? With this question we want to investigate to which extent the specification of feature contexts impairs the usual programming workflow developers are accustomed to. To answer this question, we listed all unique feature contexts necessary to reproduce each pattern in Section 6.3 for each variability-related code change pattern identified by Stănciulescu et al. [SBWW16]. We compare the amount of necessary feature contexts with the amount of unique desired mappings for each pattern. Further, we consider the number of unique variants (i.e., clones) that have to be edited. Editing multiple variants is necessary when multiple desired feature mappings contradict each other or no variant exists implementing all contexts at once.

https://doi.org/10.24355/dbbs.084-202002271120-0

We begin by inspecting patterns related to pure insertions. An overview of these patterns can be found in Table 6.2. The patterns *AddNormalCode*, *AddIfdefElse*, *AddIfdef*, and its repetition *AddIfdef*^{n} cover pure insertions of new code with (or without) desired feature mappings as indicated by the *involved derivations* column. *AddNormalCode* and *AddIfdef* require editing a single variant with a single feature context only. Considering the n-fold repetition *AddIfdef*^{n} of pattern *AddIfdef* separately is useful as it illustrates the opportunity to reuse the same feature context when it is required multiple times. Some of the desired feature mappings $c_1, \ldots, c_n$ may be equal and can be inferred with a single switch of the feature context only. However, this does not mean that developers will always perform edits such that they have to switch the feature context as rarely as possible. As finding solutions to (technical) problems and experimenting thereby are also part of programming, the actual number of context switches might be even higher than n, just as preprocessor annotations can switch multiple times during a single programming session.

Furthermore, *AddIfdef*^{n} shows the potential smaller amount of variants that have to be edited. If multiple different feature contexts are all valid for a single variant's configuration, all edits can be made in that variant. The same conclusions as for *AddIfdef*^{n} apply to repetitions of *AddNormalCode*. *AddIfdefElse* introduces two contradicting feature mappings due to the `#else` statement. Thus, we have to edit two variants with the respective feature context (i.e., the condition and its negation). As variants of product lines always implement a specific unique configuration each, a single variant can never implement contradicting features or feature interactions. Hence, we already reached the minimum number of variants to edit for *AddIfdefElse* without using artificial exploits as explained in Section 4.4. As the feature contexts equal the desired feature mapping for each pure insertion pattern (i.e., *AddNormalCode*, *AddIfdefElse*, *AddIfdef*, and *AddIfdef*^{n}), also their count equals. Except for *AddIfdef*^{n} for which potentially fewer context switches are necessary, the necessary amount of different feature contexts is the same as for preprocessor annotations to reproduce the patterns.

The code replacement patterns *AddIfdefWrapElse* and *AddIfdefWrapThen*, also summarised in Table 6.2, both require to delete an artefact and insert another one. Both patterns can be reproduced in a single variant by replacing an artefact. Here, two new feature mappings can be inferred with specifying just a single feature context. This is possible because this feature context is interpreted in a different way by each of the two consecutive different edits. Furthermore, both patterns can reproduced within just a single variant.

The two code-removing patterns *RemNormalCode* and *RemIfdef* summarised in Table 6.3 both potentially allow the feature context to be empty. For reproducing *RemIfdef*, the feature context can always be undefined (i.e., *null*). Thus, switching the context is necessary at most one time, namely when already another context is set that has to be reset manually. For reproducing *RemNormalCode*, the feature context can be undefined if the artefact to delete already has a feature mapping, otherwise it has to be *true*. Analogously to *RemIfdef*, *RemNormalCode* also requires switching the feature context thereby at most one time for each variant. As this pattern covers removing `#ifdef` with or without subsequent `#else`, one or two variants have to be

https://doi.org/10.24355/dbbs.084-202002271120-0

edited. As for pattern *AddIfdefElse*, two variants have to be edited if an `#else` block is present because both annotations are contradictory.

Finally, we consider the remaining four annotation-change patterns summarised in Table 6.4. Both, *WrapCode* (i.e., adding an annotation) and *UnwrapCode* (i.e., removing an annotation) require a single edit under a single feature context in a single variant. As *MoveElse* is a special case of *ChangePC*, we do not consider it by itself in detail. It only requires editing two different variants as again two contradicting feature mappings are involved. In *ChangePC*, the feature mapping of an artefact is changed to another arbitrary formula. Reproducing this pattern requires to delete the artefact and reinsert it with the feature context being the new mapping. As no context (i.e., *null*) has to be specified for the first step, the context has to be switched at most two times in total, similar to the code-removing patterns *RemNormalCode* and *RemIfdef*. If a variant exists whose configuration satisfies both, old and new mapping, all necessary edits can be made in just that single variant. If no such variant exists, we need to edit two variants. This can happen especially when both mappings are contradictory, as in *MoveElse*. We aim to provide custom utilities to allow intuitive manual changes of feature mappings because deleting and reinserting an artefact to remap it, is an artificial procedure in programming.

In general, we consider editing multiple variants at once to be unlikely for clone-and-own development. Patterns such as *AddIfdefElse* are natural for software product-line engineering but ineligible for variant-focused development due to their contradicting mappings. Empirical evidence for our hypothesis is given by the case study on Marlin performed by Stănciulescu et al. [SBWW16] and summarised in Table 6.1: Inserting or deleting artefacts with two contradicting feature mappings at once (*AddIfdefElse*, *RemIfdef*, *MoveElse*) is generally a more scarce edit than editing just a single mapping (*AddIfdef*, *AddNormalCode*, *RemNormalCode*) even in product-line development. Only 1.48% of all patches classified by *AddIfdefElse* could only be classified as such and the same is true for only 4.49% for pattern *RemIfdef*, which probably still contains patches without conflicting feature mappings. The pattern *MoveElse* is very rare at all.

We assume that it is more intuitive and natural for variant-oriented development (such as clone-and-own) to focus on implementing features that are present rather than solutions for absent features (i.e., negations of features). Usually, different developers are responsible for individual clones [AJB+14, DRB+13, LnBC16, RCC13, SSW15] rather than features. In that sense, each developer is primarily (but not exclusively) interested in developing his own variant rather than the others. Hence, we suppose patterns with inherent negated feature mappings to be implemented in clone-and-own development simply with different features. The following code snippet shows an example of an adapted version of *AddIfdefElse* that we suppose to be more likely and intuitive for clone-and-own development:

https://doi.org/10.24355/dbbs.084-202002271120-0

```
+ #ifdef ULTRA_LCD
+   lcd_setalertstatuspgm(lcd_msg);
+ #endif

+ #ifdef ALERT
+   alertstatuspgm(msg);
+ #endif
```

Both features `ULTRA_LCD` and `ALERT` may still be exclusive to each other and thereby preserve the semantics of *AddIfdefElse*. This contradiction can either be defined explicitly in the feature model or implicitly through the set of actually implemented configurations (e.g., both features are never present in the same variant at the same time). In that case, the first three lines belonging to feature `ULTRA_LCD` would be implemented in another variant than the other three lines mapped to `ALERT`.

In conclusion, we see that a minimal amount of feature context switches is necessary to reproduce the patterns. For all patterns except *ChangePC* and *MoveElse*, the count of necessary different feature contexts is equal or even smaller than the number of different target feature mappings. Our specialised derivation functions for different types of edits even allow inferring two desired feature mappings with just a single feature context for some patterns. Reproducing *MoveElse* and *ChangePC* is not amenable to our derivation because deriving feature mappings without edits is out of scope of our derivation. We also minimise the amount of different variants to edit as only multiple variants have to be edited for contradicting feature mappings or when no variants implement different (non-alternative) features at the same time.

6.4.2 RQ 2 – Feature Context Complexity

Our second research question is: How complex has the feature context to be in comparison to the desired feature mapping? We want the feature context specification to be as intuitive for the user as possible. This does not only increase its acceptance for workflow integration but also reduces the chance for ill-formed specifications. As for our first research question, we consider all unique feature contexts necessary to reproduce each pattern in Section 6.3 for each variability-related code change pattern identified by Stănciulescu et al. [SBWW16].

All of the patterns *AddNormalCode*, *AddIfdefElse*, *AddNormalCode*, *AddIfdef*, and its repetition $AddIfdef^n$ can be reproduced in our clone-and-own by setting the feature context to exactly the desired feature mapping for the artefact to edit. These are the patterns related to pure insertion (i.e., without further changes such as for replacements) of artefacts. We hypothesise the feature context to not impair development more than specifying preprocessor annotations for code insertions due to the formulas being identical. For the same reason, we consider the application of our feature context for code insertions in real software development to be comprehensible for developers in general. Potential initial unfamiliarity and hurdles due to the accessibility of the feature context in an IDE may arise but are not related to the patterns themselves but the feature context adaption in general.

https://doi.org/10.24355/dbbs.084-202002271120-0

For the code replacement patterns *AddIfdefWrapElse* and *AddIfdefWrapThen*, two new feature mappings can be inferred with specifying just a single feature context. The inserted artefact is mapped to the feature context φ, while the replaced artefact is mapped to the negated feature context $\neg\varphi$. We hypothesise this to be intuitive as the old artefact *does not belong to feature* φ *anymore* and the new artefact replaces it. However, we suppose distinguishing the patterns *AddIfdefWrapElse* and *AddIfdefWrapThen* to be potentially confusing for developers. While we assume replacing existing code to be intuitive, getting the negation correct when reproducing *AddIfdefWrapThen* can become intricate. We hypothesise pattern *AddIfdefWrapThen* (requiring the negated feature context) to be rather rare in clone-and-own development and that instead *AddIfdefWrapElse* is used with another convenient feature instead because *AddIfdefWrapElse* covers the case of replacing a formerly general solution (e.g., artefact) with a special case for a specific feature. In contrast, a formerly general solution is identified to be actually a special case when pattern *AddIfdefWrapThen* is applied which indicates the detection of erroneous design or a bug. Our hypothesis is further substantiated by the occurrences of the patterns in product-line development given in Table 6.1 as determined for Marlin by Stănciulescu et al. [SBWW16]. The pattern *AddIfdefWrapElse* is about three times more frequent than *AddIfdefWrapThen* in terms of overall matching patterns (*#Multi*) and nearly 6 times more frequent for exact classification of patches (*#Only*).

Removing mapped artefacts entirely is especially easy with our derivation. For both *RemNormalCode* and *RemIfdef*, *true* is always a possible feature context to choose. For *RemIfdef*, even *null* is always a possible feature context. When the deleted artefact is already mapped, *null* is also a valid feature context for reproducing *RemNormalCode*. However, no feature context other than *true* or *null* is allowed to be specified because the deleted artefact would be mapped to $\neg A \wedge \neg B$, where A is its old mapping and B the feature context. Specifying the feature context correctly here, might become confusing.

Reproducing the annotation-change patterns summarised in Table 6.4 is more intricate than the previous ones. To assign a feature mapping to a previously unmapped artefact for pattern *WrapCode* (i.e., mapped to *true* or *null* before), we have to remove it from all variants that are not allowed to contain it for a given target feature mapping m. We can achieve this by deleting the artefact with the feature context being the negated target feature mapping $\neg m$. Although these steps are reasonable from a logical perspective, we suppose them to be unintuitive when it comes to actual software development. Especially, the necessity to edit another variant to change a feature mapping of an artefact being present in a source variant can be confusing. Dedicated tool support, can enable changing feature mappings manually. Contrary, we consider reproducing *UnwrapCode* to be very amenable to our workflow and to occur as an occasional side effect during clone-and-own development. Inserting the target artefact under the desired new feature mapping (e.g., *true* in this case) is sufficient. Nevertheless, a mechanism for manual mapping changes is adequate here, too. As for the previous RQ 1, *MoveElse* is again a special case of *ChangePC* as shown in Table 6.4. Whereas *MoveElse* handles a mapping change from the concrete features $\neg U$ to U only, *ChangePC* covers arbitrary changes of

https://doi.org/10.24355/dbbs.084-202002271120-0

feature mappings. We can reproduce *ChangePC* in two steps: First, we have to delete the affected artefacts from all variants. Therefore, no feature context is necessary at all (i.e., *null* is sufficient but *true* also works). Second, the artefact has to be reinserted under the new presence condition. Although this workflow is not very pleasant to software development, we suppose this to be the easiest workflow possible when using our derivation because it only only springs into action upon edits that have to be introduced artificially here.

The most common change patterns by far are *AddNormalCode* and *RemNormalCode* according to Table 6.1. For reproducing both patterns, it is often unnecessary to specify any feature context at all due to our AST propagation introduced in Chapter 3. When artefacts are inserted into a propagating scope (e.g., a class or function definition) that already has the desired mapping, no feature context has to be specified for *AddNormalCode*. The same applies for pattern *RemNormalCode*. Thus, setting the feature context to *null* is valid to reproduce both patterns when applied inside appropriate scopes.

6.4.3 RQ 3 – Comparison to VTS

Our third and last research question is: How does our feature mapping derivation compete with the projectional editor VTS by Stănciulescu et al. considering the previous two research questions? We compare our derivation with VTS because both methods exhibit a derivation for feature mappings from a user-specified formula (i.e., feature context and ambition). First, we discuss the general differences and commonalities of VTS with or derivation. Second, we examine the differences between ambition and feature context in-depth for each code change pattern. Third, we conclude our results and address problems and notes pointed out by Stănciulescu et al. in their discussion [SBWW16].

6.4.3.1 General Differences

We begin by identifying general commonalities and differences in our feature mapping derivation and that from VTS. Both derivations are based upon an intention specified by developers upon edits they make: These are the *feature context* in our derivation and the *ambition* in VTS. Albeit this similarity, VTS and our clone-and-own enhancement have different motivations and therefore different goals as shown in Figure 6.1 on Page 77. Whereas we want to enhance clone-and-own development with product-line concepts to better trace commonalities and differences of variants, Stănciulescu et al. constructed VTS to ease software product-line development. Therefore, VTS allows developers to edit views (i.e., partial variants) instead of the whole product line at once. Upon reintegration to the central product-line repository, the changes in the partial variant are assigned a feature mapping depending on the edit and the specified ambition. To retrieve a view with VTS, a propositional formula, the *projection*, has to be specified. Albeit, our derivation does not exhibit a similar concept explicitly, a suitable projection is chosen implicitly for some patterns by selecting convenient clones to edit.

One important difference between VTS and our derivation is the handling of absent feature mappings. As VTS is a product-line editor, feature mappings are always

https://doi.org/10.24355/dbbs.084-202002271120-0

given. If no mapping is defined explicitly for an artefact (i.e., it is not surrounded by preprocessor annotations), then it is mapped to *true* because it is always present. In contrast, we work on clones without any initial domain knowledge. Therefore, not a single artefact is mapped to a feature before adopting our derivation to clone-and-own software (i.e. all mappings are *null*). Thus, we explicitly deal with absent feature mappings while VTS does not.

Furthermore, our derivation relies on the crucial but still missing subsequent synchronisation of variant for reproducing some of the code change patterns. Whereas VTS synchronises the edits to the repository with the *put* function, we do not have a concept for synchronising variants yet. Synchronising artefact edits with adherent (new or changed) feature mappings is out of scope of this thesis but an important necessary future work. Nevertheless, we have thoroughly taken care of inferring feature mappings that correctly describe the presence (or absence) of artefacts in variants when reproducing the patterns. It is yet an open topic though, how to merge different feature mappings specified differently across variants.

6.4.3.2 Ambition vs. Feature Context

In the following, we compare the ambition of VTS with our feature context. Therefore, we summarise and reflect which propositional formulas have to specified for the user for both systems for each code editing pattern.

To reproduce code-adding patterns summarised in Table 6.2, VTS almost exclusively requires the projection to be *true*. Only for *AddIfdefElse* and *AddNormalCode* different projections are necessary. For adding the `#else` block in *AddIfdefElse* the projection $\neg U$ has to be used. Inserting code into an annotated block (case (2) of *AddNormalCode*) can be done with the projection being that annotation. Using the projection *true* discards the benefits of projectional editing. The entire product line is checked out from the central repository and has to be edited at once. Contrary, in clone-and-own development as we target it, one always has to work with variants (i.e., projections). This requires to choose suitable variants for implementing specific features manually. We do not consider this to be an issue, as developers agreed on the features implemented in a (or their) variant [FLLHE15, LFLHE15, LLHE17] according to our assumptions in Section 3.1. Hence, developers know whether they can implement a requested feature in their variant or not. Managing change or feature requests is already common in industry and implemented for example via ticketing systems or issues (e.g., on GitHub[7]) [JBAC15, KDO14, DRB+13].

Our feature contexts equal the ambitions of VTS for all code-adding patterns except *AddNormalCode*. Therefore, both methods exhibit the same complexity for developers to specify their intention. Instead of specifying an ambition for pattern *AddNormalCode* in VTS, a projection of the feature to extend can be edited. This role-switch of projection and ambition could potentially become confusing for developers but also bears the opportunity for editing a projection instead of the entire product line as necessary for the other patterns. Furthermore, the same amount of variants has to be edited for all code-adding patterns except $AddIfdef^n$ for which editing potentially fewer variants can be sufficient.

[7] All issues in Marlin opened before January 17, 2020: https://github.com/MarlinFirmware/Marlin/issues?utf8=%E2%9C%93&q=is%3Aissue+created%3A%3C2020-01-17+

https://doi.org/10.24355/dbbs.084-202002271120-0

The code-removing patterns summarised in Table 6.3 are reproduced in VTS differently than with our derivation. For *RemNormalCode* and *RemIfdef* projections and ambitions are the same in VTS: Both describe the feature or feature interaction from which artefacts should be deleted. The feature context in our derivation has to be *true* or *null* to reproduce those patterns and thereby is independent from the actual feature. Thus, our derivation is able to produce the desired results even without requiring a feature context upon deletions. To avoid confusion on when the feature context is allowed to be omitted and when not, we recommend to set it to *true* for deletions.

The annotation-change patterns summarised in Table 6.4 are intricate for both VTS and our derivation. Especially *ChangePC*, the most general of the four patterns, is not amenable to both workflows. Using projections to reproduce this pattern in VTS only obfuscates the actual task, while our derivation is not applicable without artificial non-operational code changes. For VTS, Stănciulescu et al. suggest using a non-projectional edit [SBWW16, p. 329]. The only advantage of our derivation for this pattern are the more simple feature contexts (*null* instead of $\neg\,\varphi_1$, and φ_2 instead of $\neg\,\varphi_2$) and the potential to reproduce the pattern in just a single variant if its configuration satisfies source and target feature mapping. Similar to *ChangePC*, the pattern *MoveElse* is also poorly supported by both VTS and our derivation. *WrapCode* as well as *UnwrapCode* require the ambition to be a negation, whereas our feature context is only a negation for *WrapCode*. Stănciulescu et al. recommend implementing more specialised operations for *UnwrapCode* and *MoveElse* in a text editor or IDE [SBWW16, p. 331] as we also planned to do as future work in Section 6.4.1.

In general, ambition and feature context are nearly the same for code-adding patterns but differ greatly for code-removing and annotation-change patterns. Especially for code removing patterns, we do not require negations and need less negations for annotation-change patterns. Explicitly considering an absent feature context is an advantage of our derivation that allows edits to be reasonably mapped even without a feature context being specified for some patterns. Furthermore, the feature- and annotation-independent feature context *true* is often viable. To render the context being *true* reasonable to developers, *true* could serve as a *default* feature context. Moreover, reproducing the patterns with our derivation always requires editing at most the same amount of variants (or views respectively) as VTS needs.

6.4.3.3 Conclusions

Although reproducing some patterns required yet unintuitive solutions, our derivation is able to reproduce all of the presented patterns. Reasonably, Stănciulescu et al. point out that *"the edit patterns should not be seen as the edit operations a developer would use when using a variation control system"* [SBWW16, p. 331]. Similarly, we suggest the same for clone-and-own development: While most patterns describe reasonable code changes for software development in general, some of them are only suitable for product-line engineering directly. However, support for pattern *ChangePC* and thereby the other annotation-change patterns is a mandatory requirement for repairing wrong feature mappings or just updating them. Therefore, another dedicated mechanism is required that is perhaps best supported as an explicit functionality in an IDE.

https://doi.org/10.24355/dbbs.084-202002271120-0

We suppose specifying a feature context to be rather intuitive in general, especially for insertions, but agree with Stănciulescu et al. who claim that some mental effort is required in understanding what projection, ambition, and thereby also feature context mean [SBWW16, p. 331]. In their experience though, applying projection and ambition correctly was straightforward most of the time but not when multiple commits were necessary [SBWW16, p. 331]. In the clone-and-own scenario, the amount of edits can potentially be smaller than the number of necessary commits in VTS because we can switch the feature context without requiring a separate synchronisation step beforehand. Confusion may arise when deleting artefacts with our derivation. Developers have to differentiate whether they want to replace an artefact in their clone, or want to delete it from the entire software (i.e., all clones). As in both cases, a single variant is edited only, we admit this distinction to be potentially confusing despite it being necessary and reasonable from a technical point of view.

The inspected patterns do not cover moves of artefacts because the alignment (e.g., order of lines of code) of artefacts is not considered. Therefore, we did not require our derivations $\mathscr{F}_{\text{move}}$ and $\mathscr{F}_{\text{update}}$ to reproduce any of the patterns. However, we do not consider this to be crucial as both behave similar to $\mathscr{F}_{\text{insert}}$ with the difference that they can incorporate the old feature mapping of an artefact (i.e., the mapping before the edit) which is not present for insertions. Hence, we suppose that some patterns could even be reproduced in more simple ways when considering the alignment of artefacts. For example, pattern *ChangePC* could be reproduced with a single update operation if the affected line of code is updated instead of just remapped.

As views in VTS are generated by discarding all preprocessor annotations contradicting the specified projection, many orthogonal features still remain in the view although they are unrelated to the feature of interest. For instance, a view obtained with the projection A, where A is a feature, still contains code belonging to unrelated features (e.g., B, C, ...) if there are no constraints between those features. As clones are implementations of configurations of a feature model in our targeted clone-and-own scenario, these clones can also be considered as views. Thereby, clones are likely to contain less code than views obtained in VTS because their configuration explicitly deselects certain features that may be unrelated to the feature context.

Finally, we address a limitation of VTS presented by Stănciulescu et al. themselves stated as future work: *"How to handle the cases when an ambition is weaker than the projection?"* [SBWW16, p. 331]. In VTS, the ambition always has to be stronger than the projection as the projection itself is always part of the ambition internally as described in Section 6.2. For VTS, a weaker ambition could be desirable when an edit should not be executed on the edited view but affect other variants as well, for example when fixing a bug. This directly matches our targeted clone-and-own scenario. As in our scenario, the feature context (equivalent to ambition) is always weaker than the configuration (equivalent to the projection) we directly address this question. The motivation described by Stănciulescu et al. directly matches our clone-and-own scenario.

https://doi.org/10.24355/dbbs.084-202002271120-0

6.5 Threats to Validity

In this section, we depict possible threats to validity of our study. As we reuse the study by Stănciulescu et al. [SBWW16] for detection of variability-related code editing patterns in software product-line development, we inherit their possible threats to validity. In the following, we distinguish between threats to internal and external validity.

6.5.1 Internal Validity

We may have introduced a bias in considering variability-related code editing patterns not from clone-and-own but from software product-line development. Nevertheless, as we consider clones to be variants, which in turn are projections of a product line, we claim the considered edit patterns to be superset of variability-related editing patterns in variants as already discussed in Section 6.2.

We further inherit the following threats to internal validity from reusing the study by Stănciulescu et al. [SBWW16]. They admit that they *"might have introduced bias when identifying the edit operations"* [SBWW16, p. 331]. Bias could be introduced from the procedure of identifying the patterns and from subject system Marlin. To guarantee completeness of patterns for edits in Marlin's history up to a certain commit, all of the 5,640 patches until then were analysed in a systematic way. Iteratively, all patches were analysed and validated to be captured by at least one pattern. Stănciulescu et al. cross-checked the patterns with 34,018 patches from Busybox, another preprocessor-based software product line. In Busybox, 99.27% of the patches could be classified by at least one of the identified patterns. The examined version of Marlin contains about 40,000 lines of code with over 140 features in 187 source files developed in 3,747 commits by about 49 developers, rendering it a viable candidate for analysis. Busybox has about 175,000 lines of code emerged from 13,700 commits at the investigated commit wherefore we consider it to be a convenient control project.

6.5.2 External Validity

One major threat to the external validity of our result is the yet missing concept on how to synchronise feature mappings across variants which is out of scope of this thesis. Synchronising newly introduced feature mappings (changing a mapping of an artefact previously mapped to *null*) is unproblematic if the mapped artefact is already present in the variants to synchronise. However, reproducing some patterns relies on a proper synchronisation of different mappings for the same artefact specified in different variants. For instance, the mappings *true* and U have to be updated or merged to *true* as necessary for pattern *UnwrapCode* in Section 6.3.3. Whenever synchronisation is an issue for reproducing patterns, we explicitly mention that. Further, we focused on choosing reasonable mapping deductions for the future variant synchronisation.

A further yet implicit assumption is that upon pattern reproduction, our derivation algorithm always detects insertions and deletions of code fragments correctly as such. Since edit classification in our algorithm is extensible, we could always choose to implement the same change classification as Stănciulescu et al. for VTS [SBWW16] to

https://doi.org/10.24355/dbbs.084-202002271120-0

obtain the same line-based diffs. Using line-based diffs is possible for our derivation because they can be represented as ASTs of depth 1 in which all lines are represented by top-level nodes.

We further inherit the following threats to external validity from reusing the study by Stănciulescu et al. [SBWW16]. First, Stănciulescu et al. claim the identified patterns to be general enough because some of them were already identified the same in previous work [DvDP16, JBAC15, PGT+13]. By composing the identified patterns, users can edit code the same way as in a default editing model as assumed by Stănciulescu et al. [SBWW16, p- 331]. Potentially more complex patterns might stay undetected but were not necessary for the two large projects Marlin and Busybox.

6.6 Summary

In this chapter, we evaluated the applicability of our feature mapping derivation for real-world software development. Therefore, we showed how code changes in the history of the product line Marlin can be reproduced in our clone-and-own scenario with adherent feature mapping changes when using our derivation. We answered three research questions concerning the count of necessary feature context switches, the complexity of the feature context, and the comparison to the feature mapping derivation in VTS by Stănciulescu et al. [SBWW16] when reproducing these changes.

To replay variability-related real-world code changes, we reused the study conducted by Stănciulescu et al. [SBWW16]. They identified a set of variability-related code change patterns for preprocessor-based software product lines that are able to describe all code changes in the history of the printer firmware Marlin [vdZ] up to a certain commit. They used these patterns to evaluate their own projectional product-line editor VTS. These patterns can be roughly classified in patterns adding code, removing code, or patterns that solely change annotations (i.e., feature mappings). By projecting these patterns to our clone-and-own scenario, we obtained a superset of possible variability operations for it.

Important to mention is that synchronising variants is yet an open topic and a necessary future work. Without it, we indeed obtain correct feature mappings but are not able to synchronise them to other variants. This is in turn required to reproduce some of the patterns and to make use of our derivation in general. We will discuss this further in Chapter 9.

We showed that all variability-related code editing patterns can be reproduced with our feature mapping derivation in our targeted clone-and-own scenario. While not all patterns are amenable to our workflow, most of them require simple feature contexts (in terms deviation from the desired mapping) and a minimal amount of different variants to edit. We identified some patterns to be reproducible even without specifying a feature context (i.e., setting it to *null*) or setting it just to *true*. To change feature mappings themselves, artificial non-operational code changes have to be made such that our derivation can spring in to action. When implementing our derivation, a mechanism for directly changing a mapping without code changes is required.

As Stănciulescu et al. already pointed out, it is yet important to consider that the patterns do not reflect the edit operations a developer would use in other scenarios,

https://doi.org/10.24355/dbbs.084-202002271120-0

such as projectional editing or clone-and-own [SBWW16, p. 331]. We found our derivation to be as powerful as the feature mapping derivation of VTS by Stănciulescu et al. [SBWW16], while requiring slightly more simple feature contexts. In general, both methods bear the same complexity exhibit analogous workflows for most of the patterns.

https://doi.org/10.24355/dbbs.084-202002271120-0

7. Related Work

In this chapter, we give an overview on relevant other research for this thesis. Our feature mapping derivations is settled in-between software product-line research and clone-and-own software development. As the latter is more an observable phenomenon than an actual research topic we do not examine it further in this chapter. This thesis is part of the *VariantSync* project. Previous work on semi-automated feature mapping derivation on line-based mappings [Son18] assumes that each line's mapping depends on the mapping of the previous line although there is no evidence for this assumption as shown in Listing 3.2 on Page 17. In the future, we will extend other preliminary work on variant synchronisation [Pfo15] in the prototype tool [PTS+16] for line-based feature mappings.

We begin with summarising state-of-the-art in software product-line research in Section 7.1. In Section 7.2, we give an overview on variation control system. Afterwards, we sum up techniques for recovering feature mappings from existing code in Section 7.3. In Section 7.4, we picture other clone management techniques. We end by summarising existing work on tree differencing in Section 7.5.

7.1 Software Product Lines

As opposed to ad-hoc programming, software product-line engineering distinguishes between *domain* and *application engineering* [ABKS13, PBvdL05]. In domain engineering, a set of desired features of the software is identified. These features are the main development artefacts, instead of variants like in clone-and-own. In application engineering, these features are composed to variants by a dedicated variation mechanism, such as preprocessors or with plugin architectures [ABKS13, CE00, SvGB05]. Variability models, such as feature models, can be used to describe features and their valid combinations [CE00, Bat05]. We use feature models to describe all possible variants. Although only a subset of those is actually implemented in our clone-and-own scenario, the feature model thereby describes valid interactions between features and which future variants can emerge and which cannot. For implementing small partial prototypes of our work, we used the FeatureIDE library [KPK+17] that allows

https://doi.org/10.24355/dbbs.084-202002271120-0

expressing formulas of propositional calculus as well as describing feature models. Additionally, we used the graphical editors of the FeatureIDE plugin [MTS+17] for the Eclipse IDE [W+04] to create feature modelsfor our small prototypes.

Generally, there are two ways for specifying feature to code mappings necessary for application engineering: the *compositional* and the *annotative* approach [KAK08]. The compositional approach aims at modularising the software to make it extensible [GYK01]. Existing implementations thereof are component technologies [wGM02] or specialised architectures such as aspects [KLM+97], frameworks [JF88], mixin layers [SB02], multi-dimensional separation of concerns [TOHS99], or AHEAD [BSR04]. Feature mappings are specified by implementing each feature and each feature interaction in a separate module. In contrast, the annotative approach traces features by annotating the source code of a single (monolithic) code base, also known as 150% model.

Feature mappings that do not violate syntax are known as disciplined annotations (i.e., composing the mapped implementation artefacts can never lead to syntactically ill-formed artefacts for any variant). Experiments conducted to determine the necessity of discipline [LKA11, MRB+17, SLSA13] reveal the benefits for such constraints in industrial practice. We implemented disciplined annotations via abstract syntax trees as done in CIDE [KAK08] by Kästner et al. CIDE is a plugin for Eclipse that allows mapping Java code to features manually. Its name stems from it visualising feature mappings by colouring the affected code fragments.

7.2 Variation Control Systems

Variation control systems [LBG17, LELH16, SBWW16] and filtered SPLs [SW16] are hybrid variability managing solutions dealing with variability on the level of features but allow editing software product lines by editing (partial) variants of them. Usually, a projection of the product line is checked out from a central repository, edited, and re-integrated with an analogous workflow to version control systems, such as SVN or Git. Conradi and Westfechtel proposed matrices to manage to organise the version space instead of parallel branches in version control systems [CW98]. We extensively inspected the projectional product-line editor VTS by Stănciulescu et al. [SBWW16] and compared it to our approach in Chapter 6. Upon checkout, a partial variant is obtained by specifying a projection, a propositional formula specifying the feature to edit. Similar to our feature context, developers have to specify an ambition (also a propositional formula over the set of features) to describe the feature they worked on. However, the ambition has to be specified at commit, while our feature context is be specified while editing. Thereby, we enable flexibly changing the feature context fast in contrast to transaction based solutions [LELH16, SBWW16]. Although feature context specification requires an IDE, we still support development outside of an IDE when no feature context has to be specified while preserving and incorporating existing feature mappings.

ECCO by Linsbauer et al. [FLLHE15, LLHE17] stores the software in a central repository and lets users receive and submit variants. Upon commits, developers have to specify on which features they worked on. ECCO derives feature mappings from the variants developers submit and and refines those heuristically along

https://doi.org/10.24355/dbbs.084-202002271120-0

multiple commit cycles. In contrast, our approach uses an explicit mapping where developers have full control over feature locations as in CIDE [KAK08] or FeatureMapper [HKW08]. Both tools enable mapping source code artefacts to features. FeatureMapper allows feature expressions for mappings — as we do — and operates model-driven, such that not source code but model elements of the target program are annotated. Furthermore, ECCO resolves feature location uncertainty in a lazy manner, i.e., not before its necessary. As we want to be able to propagate code and mapping changes along variants at any time, we cannot afford uncertainty of feature locations. Instead, we resolve uncertainties at commit stage as these would spread across all variants and merges into the edited variant would be hindered.

7.3 Feature Mapping Recovery Techniques

Feature mapping recovery techniques [XXJ12, RC13, DRGP13, AGA13, KGP13, WKP15]. retroactively detect feature mappings in existing software. Thereby, they can be useful for our clone-and-own enhancement for identifying initial feature mappings in clones and a later migration to a software product line. Roughly, two steps are involved in migrating existing code to a software product line as classified by Martinez et al. [MZB+15]: variability or family mining [KDO14, WSSS16] (i.e., detecting variability), and the actual migration into an integrated platform [FMS+17, KFBA09, LC13]. Variability mining tools, such as LEADT by Kästner et al. [KDO14] analyse semantic dependencies of program elements to identify relationships between features. Developers have to specify initial seeds (i.e., manually map some of the artefacts). Through type-checks (e.g., detecting references of mapped code to unmapped functions or classes), neighbourhood analysis (e.g., statements belonging to the same function), and ontological analysis (i.e., checking for similarity in names), dependencies between program elements can be reverse-engineered to dependencies in features. Whereas feature recovery tools in general require numerous developer decisions [FMS+17, FLLHE15, KDO14, KKK13, LLHE17, MZB+15, RCC13, ZHP+14], fully automatic techniques [FLLHE15, LLHE17, WSSS16, ZHP+14] suffer from *unintentional divergence* [KKK13, SL14]. The actual migration can either be done stepwise (i.e., as a series of variant-preserving refactorings [FMS+17]) or in a single step, usually referred to as *big-bang migration*.

To avoid the uncertainties on recovered feature mappings of variability mining techniques, we enable developers to gradually introduce feature mappings at will without impairing ongoing software development. Thereby, we also address the risks of migrations to software product lines as feature mappings can be introduced at any speed developers prefer.

7.4 Clone Management

Existing work on clone management typically considers clones as a small-scale phenomenon [Kos07, RBS13]. Clone detection can be used as a preparatory step for product-line migration [MKB09, RCC13], described in the last section. However, they are not sufficient for managing large-scale synchronisation, maintenance, and evolution of variants as we target. Lague et al. proactively prevent cloning of software with *integrated editing* support [LPM+97]. Ducasse et al. eliminate clones

https://doi.org/10.24355/dbbs.084-202002271120-0

retroactively through refactorings [DRG99]. While both techniques are well suited for single variant development, they can hardly be applied to multiple clones. Techniques for tracking the cloning history (i.e., when a clone emerged and how) and for synchronising them [dZv09, DER10, NNP+12, TBG04] suffer from the same limitation. They depict clones on a small scale, such as a few program statements or single methods. Other approaches such as computer-aided clone-and-own by Lapeña et al. [LnBC16] rank existing artefacts according to their relevance for a new variant according to natural language requirements. Rubin et al. partially address variant synchronisation [RCC13] by discussing the applicability of a set of operators to propagate features between clones upon derivation of new variants. They do not provide a concrete implementation of those however. Annotating the version history with variability information is proposed by Schmorleitz and Lämmel [SL16] to propagate changes from one variant to another by document patching. However, they do not address change propagation failures that can occur due to conflicting changes in multiple variants, technical failures, or missing context for integration of changes into another variant. Furthermore, multiple occurrences of the same failure must be solved individually with the help of the developer for each affected variant. The potential of embedded annotations for change propagation between clones — and thereby the potential usability of the concepts developed in this thesis — was evaluated by Ji et al. [JBAC15].

Opposed to the discussed works, we enhance large-scale clone-and-own development proactively and thereby deliberately consider managing multiple clones at once. As we want to impair developers' habits and workflows by a minimal amount only, we do neither prevent cloning nor eliminate clones. Instead, we want to support a more synchronised way of clone-and-own development and even target to enhance the generation of new clones.

7.5 Tree Diffing and Semantic Lifting

To implement disciplined annotations we map features directly to the AST of source code. As we derive feature mappings from edits, we developed a notion of semantic edits on AST in Chapter 3. Therefore, we studied the literature for tree matching and diffing, summarised in Table 3.2 on Page 33. In Section 3.3 on Page 23, we give a detailed comparison of the algorithms *LaDiff* by Chawathe et al. [CRGMW96], *Diff/TS* by Hashimoto and Mori [HM08], *RTED* by Pawlik and Augsten [PA11], *GumTree* by Falleri et al. [FMB+14] and the definitions by Bille [Bil05]. To detect user-level changes in the technical edit scripts computed by these algorithms, we transferred the idea of semantic lifting by Kehrer et al. [KKT11, KKT13] to tree diffs and aim to derive an algorithm for it in the future. Kehrer et al. use semantic lifting on edits on abstract syntax graphs in the context of model-driven software development [KKT11, KKOS12] exhibiting several similarities to tree diffing.

https://doi.org/10.24355/dbbs.084-202002271120-0

8. Conclusion

Despite extensive research on software product lines, managing variability in software engineering by cloning and altering software is still common practice. Avoiding duplicate implementation effort and fixing bugs consistently throughout all clones, still remain fundamental problems in many software projects. To address these problems, we enhance clone-and-own development with product-line concepts. By identifying each clone with the unique configuration of features it implements and knowing which concrete software artefacts implement those features, we enable synchronising variants successively. Thereby, developers are able to incorporate domain knowledge during programming but without obligation to only impair their development workflow by a minimal amount.

In this thesis, we proposed the first step towards semi-automated synchronisation of clones: the recording of feature mappings during software development. Opposed to variability mining techniques which try to recover those mappings retroactively, recording feature mappings immediately delivers more accurate results. Moreover, we do not require a complete feature mapping of all artefacts in all variants but rather support developers to infer feature mappings incrementally during their usual programming workflow. Thus, our approach can even be used for a slow migration towards software product lines.

To guarantee syntactical correctness upon variant synchronisation and future variant generation, we map features to nodes in ASTs instead of lines in text documents, similar to Kästner et al. [KAK08]. AST nodes propagate their mapping to their children to express syntactical dependencies, such as fields and methods requiring their enclosing class definition. We carefully defined which nodes can be mapped to features and which ones are allowed to propagate their mapping to guarantee syntactic validity with the least amount of restrictions.

In order to be able to derive feature mappings from code changes, we classified eight tree operations, the *semantic edits*, to describe changes between two ASTs. Dedicated *semantic edits* are necessary because, intuitively, classified changes on implementation artefacts do not need to correspond to alike changes in the AST.

https://doi.org/10.24355/dbbs.084-202002271120-0

Depending on its content, inserting a line of source code can even result in restructurings. We showed that existing tree diffing algorithms do not detect such semantic edit operations but only technical tree-oriented low-level changes. Therefore, we discussed on how *semantic lifting* [KKT11], a technique known from model-driven software development, can detect semantic edits in low-level edit scripts computed by state-of-the-art tree diffing algorithms.

We developed an algorithm for deriving feature mappings upon code changes by incorporating developers domain knowledge in form of a propositional formula over the set of features, called the *feature context*. Depending on developers' edits and feature context, we assign feature mappings to the edited artefacts. We developed an individual feature mapping derivation for each edit type (i.e., insertion, deletion, update, and move) and proved that they fulfil certain constraints to avoid surprising, incomprehensible, or unintuitive results.

Opposed to previous research, we notably consider the absence of a feature context (i.e., setting it to *null*). Thus, edits outside of an IDE are supported and developers do not have to specify the feature they are working on when they do not know it. Our derivations do not require the feature context to be specified and preserve existing mappings. We also gave an outlook on how artefact detection across variants can help to determine partial feature mappings for artefacts depending on the configurations of clones. Furthermore, we showed that our AST propagation and feature mapping derivation always conforms with the feature model. We demonstrated how a global feature model could be used to simplify feature mappings and indicated potential risks thereof.

We evaluated our feature mapping derivation by replaying variability-related code editing patterns, extracted from a software product line, in the clone-and-own scenario we target. Therefore, we used the code editing patterns identified by Stănciulescu et al. [SBWW16], who investigated all commits in the history of the product line Marlin [vdZ]. We compared our method with the projectional product-line variation control system VTS by Stănciulescu et al. [SBWW16] that, similarly to our derivation, exhibits a feature mapping derivation from a user-specified formula. We showed that we can reproduce every pattern with our feature mapping derivation in clone-and-own development when no ill-formed feature mappings are present. We avoid ill-formed mappings in the first place by using ASTs.

In detail, we investigated the amount of necessary feature context switches, the feature context complexity, and differences to its equivalent in VTS when reproducing the patterns. We found our feature mapping derivation to require a minimal amount of context switches for code insertion and deletion patterns with respect to the target feature mapping and VTS. Specifying the feature context is straightforward for code-insertion patterns and almost always equal to the target feature mapping of the edited artefact. For code-removing patterns, the feature context *true*, or even *null* in some cases, is sufficient. While our feature contexts are slightly more convenient (considering desired feature mappings) than the ambitions in VTS necessary for reproducing patterns concerning solely changing feature mappings, both methods are not very amenable to these patterns. As our derivation acts on code changes, it cannot be used to statically change feature mappings without artificial code changes, such as removing and reinserting an artefact. As for VTS, an additional mechanism

https://doi.org/10.24355/dbbs.084-202002271120-0

for manual feature mapping changes would be reasonable as commits classified by such patterns can often be found in the history of Marlin.

We can imagine our derivation being especially helpful in industrial practice when combined with an issue or ticketing system in which individual tasks are assigned to developers. While bugfixes cannot be traced immediately, tickets requesting extensions, changes, or new functionality are usually associated to certain features. When starting to work on a ticket or issue, developers could set the feature context to exactly that feature and start working on their variant.

https://doi.org/10.24355/dbbs.084-202002271120-0

https://doi.org/10.24355/dbbs.084-202002271120-0

9. Future Work

Finally, we summarise possible future work to extend and continue the work of this thesis. In the following, we depict each future work separately.

Variant Synchronisation

Synchronising edits and feature mappings between clones is essential for clone-and-own enhancement. We plan to extend existing work based on patches and line-based feature mappings [PTS+16]. Important for variant synchronisation is resolving merge conflicts: When feature mappings of the same artefact are changed simultaneously in different variants, we have to choose which feature mapping is the correct one, or if both have to be merged somehow.

Semantic Lifting on ASTs

Lifting technical low-level edit scripts describing changes between two trees to user-level edit script is still an open topic. In Chapter 3, we discussed the necessity of semantic lifting to reasonably detect user made changes to implementation artefacts in-depth. Existing work only recovers low-level edit scripts that do not properly reflect developers intents as shown in Section 3.3.

Dependency Detection of AST Nodes

So far, we designate nodes in the AST to propagate their feature mapping to their children when they are *hierarchically mandatory* (i.e., they cannot be removed without invalidating their children). Contrary, we do not let *hierarchically optional* nodes propagate their mapping. This may however be reasonable, when for instance a variable is defined in the expression of a condition (e.g., `if (bool b = input()) { print(b); }`) and used inside that condition. Therefore, further analyses to detect such dependencies are necessary.

https://doi.org/10.24355/dbbs.084-202002271120-0

Feature Context Backpropagation During Semantic Lifting

To correctly adapt intentions and expectations of developers, it could be useful to retroactively assign feature contexts to edits for which no feature context was specified. Consider Example 4.1.1 on Page 51. Two edits are necessary to reproduce the inspected code editing example correctly. Both have to be executed under the same feature context. When users forget to set the feature context for the first edit, this can either be intentional or by mistake. Possible ways to detect this issue and provide tool support for it could be desirable.

Repair Wrong Feature Mappings

We did not yet discuss the robustness of our concept against accidentally wrong feature contexts or mappings. If developers make edits under a wrong feature context that has to be changed later, wrong edits may already be synchronised wrongly to other variants. A mechanism for undoing synchronisations could prevent harmful bug spread. Changing feature mappings locally and retroactively should be straightforward but also requires further utility such that no artificial code changes are necessary as discussed in Section 6.4. Here the exploits presented in Section 4.4 could prove useful when used by an automatism.

Explanations on Derivations

Providing feedback on more intricate derivations such as $\mathscr{F}_{\text{delete}}$ could be helpful. Although we designed our feature mapping derivation to be as intuitive as possible, there might still occur confusing or incomprehensible cases. Developers could then request explanations on derivations if the feature mapping derivation does not act in a comprehensible way. In Section 4.2, we discussed benefits and issues for our feature mapping derivation imposed by constraints on possible configurations and feature mappings of the global feature model. Our derivation $\mathscr{F}_{\text{delete}}$ may produce feature mappings violating the feature model. This is not an issue, as this simply means, that a deleted artefact has to be deleted from every variant. Thus, its mapping can be simplified to *false*. However, this should be communicated or explained to the user to avoid unintentional or unintuitive behaviour. Therefore, existing work on explanations on feature model defects could potentially be reused [Gün17].

Enhancing Updates

Our notion of updates on AST nodes, introduced in Section 3.3.1, is yet rather primitive. We consider only changing a single node's type or value as an update. Thereby, our notion of updates does not reflect coherent changes in multiple locations, such as refactoring the class name across the entire code base. Differentiating this edit from replacing an artefact with another one is ambiguous, too. Detecting renaming of structures would enable a correct synchronisation across variants, as such a renaming operation could be executed on target variants instead of synchronising a set of local updates. This would circumnavigate the problem that not all updates may have a target in other variants and that some locations in the target variants would not get updated because they do not have a counterpart in the source variant.

https://doi.org/10.24355/dbbs.084-202002271120-0

Metric for Artefact Equivalence

To synchronise artefacts and feature mappings between variants we need a notion of when artefacts are considered to be equal. Equivalence of artefacts is dependent on their data (e.g., text) and their location. For text documents, locations can be identified by file name and line number. Alternatively, as we already use ASTs as feature mapping targets, we can reuse those to describe locations of artefacts. In contrast to an identification with line numbers, ASTs enable an order independent comparison of artefacts. For instance, methods defined in class can have arbitrary order in different variants but can still be considered equal if their name and content is identical.

Numerical Feature Support

In preprocessor-based software product lines, such as Marlin [vdZ] investigated in Chapter 6, many numerical features can be found. Usually features are two-valued, i.e., the are either selected or deselected. Unlike these, numerical features are identified by integer or float values. These features are often checked for certain values or bounds in preprocessor `#ifdef` statements and thereby form a boolean expression again (e.g., $intensity > 3$). In that sense, numerical features could be supported by considering only the logical expressions they are contained in. Elsewise, first-order logic is required to incorporate numerical values correctly. Therefore, existing work on extended feature models [BSRC10] could become useful as these are feature models extended by first-order logic.

Literal List Feature Context

Instead of specifying a propositional formula, feature contexts could be specified as a list of literals as done in CIDE [KAK08] or ECCO [FLLHE15]. In both methods, a list of those features implemented by the current artefact or edit is specified. While literals are allowed to be negated in ECCO, this is not the case for CIDE. By using the knowledge on configurations of variants, we can derive partial feature mappings as shown in Section 4.3 on Page 63. By considering only those literals specified in the feature context list for partial feature mapping deduction, we could derive the actual feature mappings with a certain imprecision as we verified in a small idealised test environment. The question whether such a feature context simplification pays off, and how we can avoid or handle produced imprecision, is subject to future work.

Feature Mapping Simplification

By detecting redundancies due to the AST propagation (investigated in Section 5.1 on Page 69) or considering constraints of the global feature model, feature mappings can be simplified. By using existing research on feature models [vRGA+15] and decision propagation [KTS+18], we could automatically simplify feature mappings or recommend their simplification to developers. However, if the feature model evolves, simplified feature mappings may no longer be correct and thus introduce variability bugs.

https://doi.org/10.24355/dbbs.084-202002271120-0

https://doi.org/10.24355/dbbs.084-202002271120-0

Semi-Automated Inference of Feature Traceability During Software Development

Master Thesis - Task Definition

30.07.2019

Student: Paul Maximilian Bittner
Supervisors: Ina Schaefer, Thomas Thüm, Tobias Pett 30.07.2019

1 Introduction

Modern software is often required in multiple variants. Naturally software development starts with just a single variant to reduce complexity and costs and because future variants are commonly unknown [1, 2]. When the need for a new variant emerges, the whole software is cloned to alter specific parts independently from the previous variant. This ad-hoc solution is known as clone-and-own [1, 3, 4, 5]. However, with growing number of variants clone-and-own becomes infeasible because synchronising changes between variants becomes confusing and tedious.

Software product lines allow managing variants by mapping implementation artefacts to features. These features are shared and reused across variants [6, 7]. However, this design requires dedicated tool support, workflow adaptations and time for careful *domain engineering* and thus is only used rarely in practice. Furthermore, with growing number of variants, a later migration from clone-and-own to a software product line becomes increasingly difficult and time-consuming. Hence, it is bears high risks and costs.

Therefore, a novel hybrid approach proposed in the research project *VariantSync* aims at synchronising clone-and-own variants with software product line technology [8]. The domain knowledge about features should be used such that each variant comprises a unique configuration of a common feature model. Software artefacts such as source code are annotated with their corresponding feature or feature interaction. Based on this information, changes can be propagated automatically.

The first step towards this automation is the introduction of feature traceability via source code annotation. During development these mappings have to updated as the code changes to allow variant synchronisation. It is yet unclear how, if, and how far code insertions and deletions can be safely mapped to features without requiring the developers expertise. Thus, beside manual source code annotation with features, we develop methods for inferring feature mappings from artefact changes. In addition to previous line-based approaches, we investigate the possibility of using knowledge about syntax and semantics of the edited source code file. Furthermore, we will evaluate if the configuration of a variant can be used as a source of information. For example only features that appear in such a configuration are viable in a feature context.

https://doi.org/10.24355/dbbs.084-202002271120-0

2 Research

The *VariantSync* project is settled in-between software product line research and clone-and-own software development. As the latter is more an observable phenomenon than an actual research topic it is not examined further.

As opposed to clone-and-own, Software Product Line Engineering (SPLE) distinguishes between *domain* and *application engineering* [6, 9]. Therefore, variants are analysed to detect common features beforehand. These features are the main development artefacts, instead of variants like in clone-and-own. Afterwards, these features are composed to variants by a dedicated variation mechanism, e.g., preprocessor annotations or plugins [6, 7, 10]. Typically, feature models are used to describe features and their valid combinations [7, 11]. In *VariantSync*, these feature models are reused to describe all possible variants, although only a subset of these are implemented. For our feature mapping we use the features given in a common feature model for all variants. Furthermore, we aim to use the feature model for analysis purposes.

Other hybrid variability managing approaches, such as Ecco [12], identify feature mappings implicitly and heuristically and refine those along multiple changes in variants. Here, the responsibility for correct feature mappings is shifted from the developer to an automation system. In contrast, our approach uses an explicit mapping where developers have full control over feature locations as in CIDE [13]. Furthermore, Ecco resolves feature location uncertainty in a lazy manner, i.e., not before its necessary. As we want to be able to propagate code and mapping changes along variants at any time, we cannot afford uncertainty of feature locations. Instead, we resolve uncertainties at commit stage as these would spread across all variants and merges into the edited variant would be hindered.

Correct feature mappings that do not violate syntax and semantics are known as disciplined annotations. Experiments conducted to determine the necessity of discipline [14, 15, 16] can reveal the benefits for such constraints.

Feature mapping recovery techniques [17, 18, 19, 20, 21, 22, 23]. are useful for a migration of a clone-and-own software system to a system managed by *VariantSync* or even a software product line. For this work, which focuses on the ongoing development with such a managed system, these methods could help at feature mapping deduction from artefact changes.

3 Concept

Besides technical challenges, mapping artefacts to features should be as easy as possible for the developer. This can become especially intricate for feature interactions. Here, propositional formulas over the set of features are necessary rather than single features. Therefore, we want to (semi-) automatically deduce such feature formulas from artefact changes in a way that is intuitive and comprehensible for the developer. Furthermore, development could become faster and safer if developers would only have to specify feature (not feature formulas) mappings. Hence, we want to derive feature formula mappings from feature mappings by using domain specific knowledge of the edited artefacts:

1. Derive a translation from feature mappings and domain-specific knowledge to feature formula mappings for source code if possible. (SHOULD)

2. Develop a heuristic for (semi-) automating feature formula mapping from artefact changes. (SHOULD)

https://doi.org/10.24355/dbbs.084-202002271120-0

4 Implementation

Straightforward line- or symbol-based mappings do not make use of knowledge about the annotated artefacts structure. For example, source code can be annotated such that syntax or semantic would be violated when a feature gets added or removed in another variant. Furthermore, mapped feature formulas should not evaluate to false under the current variants configuration. It must not be possible to annotate artefacts with such feature formulas as this would conflict the variant itself.

The *VariantSync* software should be as reusable and extensible as possible. Therefore, its functionality will be delivered as a Java library that can be used to implement tool-specific annotation editors.

Together with the aforementioned tasks, all conceptual tasks given in Section 3 MUST be implemented in the *VariantSync* software:

1. Refine and extend the existing *VariantSync* framework capabilities of line-based feature context mappings to allow manual and explicit mapping. This is necessary as a fall-back when an automatic mapping inference fails. (MUST)
2. Feature context mappings should be stable against external code changes. If a document gets changed outside of our mapping tool, the mappings should be preserved as accurately as possible. (CAN)
3. Prevent syntactically illegal feature mappings: Removing a feature must leave the program in a syntactically valid state. (CAN)
4. Prevent semantically illegal feature mappings: Removing a feature must leave the program in a semantically valid state.(CAN)
5. Prevent false feature contexts. (MUST)
6. Implement the translation from feature mappings and domain-specific knowledge to feature formula mappings for source code. (SHOULD)
7. Implement the heuristic for (semi-) automating feature formula mapping from artefact changes. (SHOULD)

5 Evaluation

In the following we describe how concept and implementation will be evaluated. To measure correctness and usability of both our feature-to-feature-formula translation and our heuristic, we need ground truth data. Therefore a real product line will be used as these already provide mappings involving feature interactions, for example with preprocessor annotations.

1. If we find an exact translation from feature mappings and domain knowledge to feature formula mappings as described in Section 3, it should be correct by construction. We will verify this by mapping the product lines source code to single features in our tool. If our tools translation of this mapping matches the original product line's mapping, our derivation works as intended. (MUST)
2. To measure the usability and efficiency of our heuristic for feature mapping support during development, two groups of students or developers could fulfil the same programming task. One group would use our assistance tool, whereas the second does not. Thereby, we would want to answer the following questions: Does the supporting heuristic behave as expected? Is it helpful or does it hinder users? (CAN)

https://doi.org/10.24355/dbbs.084-202002271120-0

5.1 Research Questions

As part of the evaluation the following research questions should be answered.

RQ 1: How far can we statically deduce feature formula mappings from feature mappings for source code? (Must)

The tedious and error-prone task of specifying feature formulas could be simplified to a certain degree or if these could be derived from more intuitive sole feature mappings.

RQ 2: To which degree can dynamic feature formula mapping deduction from artefact changes be automated? (Should)

The more precise and self-contained our feature formula deduction is, the more work and responsibility is drawn from the developer. At best, feature context mappings could be deduced without the developers interaction for smaller changes. However, even partially automated methods could support developers greatly.

References

[1] 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," pp. 532–535, 2014.

[2] L. Linsbauer, S. Fischer, R. E. Lopez-Herrejon, and A. Egyed, "Using traceability for incremental construction and evolution of software product portfolios," pp. 57–60, 2015.

[3] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An Exploratory Study of Cloning in Industrial Software Product Lines," pp. 25–34, 2013.

[4] J. Rubin, K. Czarnecki, and M. Chechik, "Managing Cloned Variants: A Framework and Experience," pp. 101–110, 2013.

[5] S. Stănciulescu, S. Schulze, and A. Wąsowski, "Forked and Integrated Variants in an Open-Source Firmware Project," pp. 151–160, Sept. 2015.

[6] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. 2013.

[7] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. 2000.

[8] T. Pfofe, T. Thüm, S. Schulze, W. Fenske, and I. Schaefer, "Synchronizing Software Variants with VariantSync," pp. 329–332, Sept. 2016.

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

[10] M. Svahnberg, J. van Gurp, and J. Bosch, "A Taxonomy of Variability Realization Techniques: Research Articles," vol. 35, pp. 705–754, July 2005.

[11] D. Batory, "Feature Models, Grammars, and Propositional Formulas," pp. 7–20, 2005.

https://doi.org/10.24355/dbbs.084-202002271120-0

[12] L. Linsbauer, A. Egyed, and R. E. Lopez-Herrejon, "A Variability Aware Configuration Management and Revision Control Platform," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, (New York, NY, USA), pp. 803–806, ACM, 2016.

[13] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines," in *ICSE*, (NY), pp. 311–320, ACM, May 2008.

[14] J. Liebig, C. Kästner, and S. Apel, "Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code," in *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, (New York, NY, USA), pp. 191–202, ACM, 2011.

[15] R. Malaquias, M. Ribeiro, R. Bonifácio, E. Monteiro, F. Medeiros, A. Garcia, and R. Gheyi, "The discipline of preprocessor-based annotations - does #ifdef tag n't #endif matter," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pp. 297–307, May 2017.

[16] S. Schulze, J. Liebig, J. Siegmund, and S. Apel, "Does the Discipline of Preprocessor Annotations Matter?: A Controlled Experiment," *SIGPLAN Not.*, vol. 49, pp. 65–74, Oct. 2013.

[17] R. Koschke, P. Frenzel, A. P. Breu, and K. Angstmann, "Extending the Reflexion Method for Consolidating Software Variants into Product Lines," vol. 17, pp. 331–366, Dec. 2009.

[18] Y. Xue, Z. Xing, and S. Jarzabek, "Feature Location in a Collection of Product Variants," pp. 145–154, 2012.

[19] J. Rubin and M. Chechik, "A Survey of Feature Location Techniques," in *Domain Engineering: Product Lines, Languages, and Conceptual Models* (I. Reinhartz-Berger, A. Sturm, T. Clark, S. Cohen, and J. Bettin, eds.), pp. 29–58, 2013.

[20] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature Location in Source Code: A Taxonomy and Survey," vol. 25, no. 1, pp. 53–95, 2013.

[21] N. Ali, Y.-G. Gueheneuc, and G. Antoniol, "Trustrace: Mining Software Repositories to Improve the Accuracy of Requirement Traceability Links," vol. 39, pp. 725–741, May 2013.

[22] H. Kagdi, M. Gethers, and D. Poshyvanyk, "Integrating Conceptual and Logical Couplings for Change Impact Analysis in Software," vol. 18, no. 5, pp. 933–969, 2013.

[23] C. Kästner, A. Dreiling, and K. Ostermann, "Variability Mining: Consistent Semiautomatic Detection of Product-Line Features," vol. 40, pp. 67–82, Jan. 2014.

https://doi.org/10.24355/dbbs.084-202002271120-0

https://doi.org/10.24355/dbbs.084-202002271120-0

Bibliography

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. 2013. (cited on Page 1, 5, 11, and 101)

[AGA13] Nasir Ali, Yann-Gael Gueheneuc, and Giuliano Antoniol. Trustrace: Mining Software Repositories to Improve the Accuracy of Requirement Traceability Links. 39(5):725–741, May 2013. (cited on Page 57 and 103)

[AJB+14] MichałAntkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Stefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. Flexible Product Line Engineering with a Virtual Platform. pages 532–535, 2014. (cited on Page 1, 2, 5, 10, 14, 15, and 91)

[AKL13] Sven Apel, Christian Kästner, and Christian Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. 39(1):63–79, January 2013. (cited on Page 22)

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. 2006. (cited on Page 17)

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. pages 7–20, 2005. (cited on Page 6 and 101)

[BCH+10] Quentin Boucher, Andreas Classen, Patrick Heymans, Arnaud Bourdoux, and Laurent Demonceau. Tag and Prune: A Pragmatic Approach to Software Product Line Implementation. pages 333–336, 2010. (cited on Page 71)

[Bil05] Philip Bille. A Survey on Tree Edit Distance and Related Problems. *Theoretical computer science*, 337(1-3):217–239, 2005. (cited on Page 2, 31, 32, 33, and 104)

[bKLW12] A. ben Fadhel, M. Kessentini, P. Langer, and M. Wimmer. Search-based detection of high-level model changes. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 212–221, Sep. 2012. (cited on Page 34)

https://doi.org/10.24355/dbbs.084-202002271120-0

[BRN+13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A Survey of Variability Modeling in Industrial Practice. pages 7:1–7:8, 2013. (cited on Page 11)

[BSR04] Don Batory, Jacob N. Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. 30(6):355–371, 2004. (cited on Page 8 and 102)

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010. (cited on Page 111)

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. 2000. (cited on Page 1, 5, 6, and 101)

[CRGMW96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change Detection in Hierarchically Structured Information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 493–504, New York, NY, USA, 1996. ACM. (cited on Page 2, 31, 32, 33, and 104)

[Cur62] H. Allen Curtis. "Chapter 2.3. McCluskey's Method". In *A New Approach to The Design of Switching Circuits. The Bell Laboratories Series*, pages 90–160. D. van Nostrand Company, Inc, 1962. (cited on Page 64)

[CW98] Reidar Conradi and Bernhard Westfechtel. Version Models for Software Configuration Management. 30(2):232–282, June 1998. (cited on Page 10 and 102)

[DER10] Ekwa Duala-Ekoko and Martin P. Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Trans. Softw. Eng. Methodol.*, 20(1), July 2010. (cited on Page 104)

[DP16] Georg Dotzler and Michael Philippsen. Move-optimized source code tree differencing. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 660–671. IEEE, 2016. (cited on Page 32)

[DRB+13] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An Exploratory Study of Cloning in Industrial Software Product Lines. pages 25–34, 2013. (cited on Page 1, 2, 9, 14, 91, and 95)

[DRG99] Stéphane Ducasse, Matthias Rieger, and Georges Golomingi. Tool support for refactoring duplicated oo code. In *Proceedings of the ECOOP'99 Workshop on Experiences in Object-Oriented Re-Engineering*. Citeseer, 1999. (cited on Page 104)

https://doi.org/10.24355/dbbs.084-202002271120-0

[DRGP13] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature Location in Source Code: A Taxonomy and Survey. 25(1):53–95, 2013. (cited on Page 57 and 103)

[DvDP16] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. FEVER: Extracting Feature-Oriented Changes from Commits. pages 85–96, 2016. (cited on Page 99)

[dZv09] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *2009 IEEE International Conference on Software Maintenance*, pages 169–178, Sep. 2009. (cited on Page 104)

[FLLHE15] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. The ECCO Tool: Extraction and Composition for Clone-and-Own. pages 665–668, 2015. (cited on Page 2, 14, 38, 95, 102, 103, and 111)

[FMB+14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014. (cited on Page 2, 31, 32, 33, and 104)

[FMS+17] Wolfram Fenske, Jens Meinicke, Sandro Schulze, Steffen Schulze, and Gunter Saake. Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line. pages 316–326, 2017. (cited on Page 1, 2, 10, 11, and 103)

[Gün17] Timo Günther. Explaining Satisfiability Queries for Software Product Lines. Master's thesis, Braunschweig, 2017. (cited on Page 110)

[GYK01] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, page 265–274, USA, 2001. IEEE Computer Society. (cited on Page 102)

[HBC+12] Patrick Heymans, Quentin Boucher, Andreas Classen, Arnaud Bourdoux, and Laurent Demonceau. A Code Tagging Approach to Software Product Line Development. 14:553–566, 2012. (cited on Page 71)

[HKW08] Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping Features to Models. pages 943–944, May 2008. Informal demonstration paper. (cited on Page 103)

[HM08] M. Hashimoto and A. Mori. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *2008 15th Working Conference on Reverse Engineering*, pages 279–288, Oct 2008. (cited on Page 2, 31, 32, 33, and 104)

https://doi.org/10.24355/dbbs.084-202002271120-0

[JBAC15] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. Maintaining Feature Traceability with Embedded Annotations. pages 61–70, 2015. (cited on Page 14, 95, 99, and 104)

[JF88] Ralph E Johnson and Brian Foote. Designing reusable classes. *Journal of object-oriented programming*, 1(2):22–35, 1988. (cited on Page 8 and 102)

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *ICSE*, pages 311–320, NY, May 2008. ACM. (cited on Page , 2, 8, 9, 16, 17, 19, 35, 63, 71, 102, 103, 105, and 111)

[KAT+09] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don Batory. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In Manuel Oriol and Bertrand Meyer, editors, *Objects, Components, Models and Patterns*, pages 175–194, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. (cited on Page 2, 22, and 35)

[KDO14] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. Variability Mining: Consistent Semiautomatic Detection of Product-Line Features. 40(1):67–82, January 2014. (cited on Page 1, 2, 10, 11, 14, 57, 95, and 103)

[KFBA09] Rainer Koschke, Pierre Frenzel, Andreas P. Breu, and Karsten Angstmann. Extending the Reflexion Method for Consolidating Software Variants into Product Lines. 17(4):331–366, December 2009. (cited on Page 1, 10, 11, 57, and 103)

[KGP13] Huzefa Kagdi, Malcom Gethers, and Denys Poshyvanyk. Integrating Conceptual and Logical Couplings for Change Impact Analysis in Software. 18(5):933–969, 2013. (cited on Page 57 and 103)

[KKK13] Benjamin Klatt, Martin Küster, and Klaus Krogmann. A Graph-Based Analysis Concept to Derive a Variation Point Design from Product Copies. pages 1–8, March 2013. (cited on Page 2 and 103)

[KKOS12] T. Kehrer, U. Kelter, M. Ohrndorf, and T. Sollbach. Understanding model evolution through semantically lifting model differences with silift. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 638–641, Sep. 2012. (cited on Page 34 and 104)

[KKT11] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. A Rule-based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 163–172, Washington, DC, USA, 2011. IEEE Computer Society. (cited on Page , 2, 13, 33, 34, 35, 104, and 106)

https://doi.org/10.24355/dbbs.084-202002271120-0

[KKT13] T. Kehrer, U. Kelter, and G. Taentzer. Consistency-preserving edit scripts in model versioning. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 191–201, Nov 2013. (cited on Page 104)

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. pages 220–242, 1997. (cited on Page 8, 9, and 102)

[Kos07] Rainer Koschke. Survey of Research on Software Clones. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. (cited on Page 103)

[KPK+17] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. FeatureIDE: Empowering Third-Party Developers. pages 42–45, 2017. (cited on Page 101)

[Kru02] Charles W. Krueger. Easing the Transition to Software Mass Customization. pages 282–293, 2002. (cited on Page 11)

[KTS+18] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. Propagating Configuration Decisions with Modal Implication Graphs. pages 898–909, May 2018. (cited on Page 62 and 111)

[LBG17] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. A classification of variation control systems. pages 49–62, 2017. (cited on Page 102)

[LC13] Miguel A. Laguna and Yania Crespo. A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring. *Science of Computer Programming*, 78(8):1010–1034, August 2013. (cited on Page 1, 10, 11, and 103)

[LELH16] Lukas Linsbauer, Alexander Egyed, and Roberto Erick Lopez-Herrejon. A Variability Aware Configuration Management and Revision Control Platform. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 803–806, New York, NY, USA, 2016. ACM. (cited on Page 65 and 102)

[LFLHE15] Lukas Linsbauer, Stefan Fischer, Roberto E. Lopez-Herrejon, and Alexander Egyed. Using traceability for incremental construction and evolution of software product portfolios. pages 57–60, 2015. (cited on Page 1, 14, and 95)

https://doi.org/10.24355/dbbs.084-202002271120-0

[LKA11] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 191–202, New York, NY, USA, 2011. ACM. (cited on Page 22 and 102)

[LLHE17] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Variability Extraction and Modeling for Product Variants. 16(4):1179–1199, October 2017. (cited on Page 2, 14, 38, 95, 102, and 103)

[LnBC16] Raúl Lapeña, Manuel Ballarin, and Carlos Cetina. Towards Clone-and-Own Support: Locating Relevant Methods in Legacy Products. pages 194–203, 2016. (cited on Page 2, 91, and 104)

[Lon] King's College London. Bluej IDE. https://www.bluej.org/. Accessed at January 06, 2020. (cited on Page , 71, and 72)

[LPM+97] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *1997 Proceedings International Conference on Software Maintenance*, pages 314–321, Oct 1997. (cited on Page 103)

[MAaL18] Willian D. F. Mendonça, Wesley K. G. Assunção, and Lukas Linsbauer. Multi-objective optimization for reverse engineering of apogames feature models. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*, SPLC '18, page 279–283, New York, NY, USA, 2018. Association for Computing Machinery. (cited on Page 11 and 64)

[MHK19] Junnosuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto. Beyond GumTree: A Hybrid Approach to Generate Edit Scripts. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, pages 550–554, Piscataway, NJ, USA, 2019. IEEE Press. (cited on Page 32)

[MKB09] Thilo Mende, Rainer Koschke, and Felix Beckwermert. An Evaluation of Code Similarity Identification for the Grow-and-Prune Model. 21(2):143–169, March 2009. (cited on Page 103)

[MRB+17] R. Malaquias, M. Ribeiro, R. Bonifácio, E. Monteiro, F. Medeiros, A. Garcia, and R. Gheyi. The Discipline of Preprocessor-Based Annotations - Does #ifdef TAG n't #endif Matter. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 297–307, May 2017. (cited on Page 102)

[MTS+17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. 2017. (cited on Page 102)

https://doi.org/10.24355/dbbs.084-202002271120-0

[MZB+15] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Bottom-Up Adoption of Software Product Lines: A Generic and Extensible Approach. pages 101–110, 2015. (cited on Page 2, 11, and 103)

[NMLT08] A. D. Nicola, T. D. Mascio, M. Lezoche, and F. Tagliano. Semantic Lifting of Business Process Models. In *2008 12th Enterprise Distributed Object Computing Conference Workshops*, pages 120–126, Sep. 2008. (cited on Page 34)

[NNP+12] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Clone Management for Evolving Software. 38(5):1008–1026, September 2012. (cited on Page 104)

[PA11] Mateusz Pawlik and Nikolaus Augsten. RTED: A Robust Algorithm for the Tree Edit Distance. *Proc. VLDB Endow.*, 5(4):334–345, December 2011. (cited on Page 2, 31, 32, 33, and 104)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* September 2005. (cited on Page 5, 11, and 101)

[Pfo15] Tristan Pfofe. Automating the Synchronization of Software Variants. Master's thesis, Magdeburg, 2015. (cited on Page 101)

[PGT+13] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wąsowski, and Paulo Borba. Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. pages 91–100, 2013. (cited on Page 99)

[Pre97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. pages 419–443, 1997. (cited on Page 9)

[PTS+16] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. Synchronizing Software Variants with VariantSync. pages 329–332, September 2016. (cited on Page 101 and 109)

[RBS13] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software Clone Detection: A Systematic Review. *Information and Software Technology*, 55(7):1165–1199, 2013. (cited on Page 103)

[RC13] Julia Rubin and Marsha Chechik. A Survey of Feature Location Techniques. In Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and Jorn Bettin, editors, *Domain Engineering: Product Lines, Languages, and Conceptual Models*, pages 29–58. 2013. (cited on Page 14, 57, and 103)

[RCC13] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Managing Cloned Variants: A Framework and Experience. pages 101–110, 2013. (cited on Page 1, 2, 9, 14, 91, 103, and 104)

https://doi.org/10.24355/dbbs.084-202002271120-0

[SB02] Yannis Smaragdakis and Don Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. 11(2):215–255, April 2002. (cited on Page 102)

[SBWW16] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. pages 323–333, October 2016. (cited on Page , 3, 38, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 91, 92, 93, 94, 96, 97, 98, 99, 100, 102, and 106)

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software: beyond object-oriented programming*. Pearson Education, 2002. (cited on Page 8)

[SL14] Thomas Schmorleiz and Ralf Lämmel. Similarity Management via History Annotation. pages 45–48. Dipartimento di Informatica Università degli Studi dell'Aquila, L'Aquila, Italy, July 2014. (cited on Page 2 and 103)

[SL16] Thomas Schmorleiz and Ralf Lämmel. Similarity Management of 'Cloned and Owned' Variants. pages 1466–1471, 2016. (cited on Page 104)

[SLSA13] Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. Does the Discipline of Preprocessor Annotations Matter?: A Controlled Experiment. *SIGPLAN Not.*, 49(3):65–74, October 2013. (cited on Page 102)

[Son18] Christopher Sontag. Recording Feature Mappings During Evolution of Cloned Variants. Master's thesis, Braunschweig, 2018. (cited on Page 2, 45, 46, and 101)

[SSW15] Stefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. Forked and Integrated Variants in an Open-Source Firmware Project. pages 151–160, September 2015. (cited on Page 1, 2, 14, and 91)

[SvGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A Taxonomy of Variability Realization Techniques: Research Articles. 35(8):705–754, July 2005. (cited on Page 101)

[SW16] Felix Schwägerl and Bernhard Westfechtel. SuperMod: Tool Support for Collaborative Filtered Model-Driven Software Product Line Engineering. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 822–827, New York, NY, USA, 2016. Association for Computing Machinery. (cited on Page 102)

[TBG04] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing Duplicated Code with Linked Editing. pages 173–180, 2004. (cited on Page 104)

https://doi.org/10.24355/dbbs.084-202002271120-0

[Thü18] Thomas Thüm. Lecture Notes – Software-Produktlinien: Konzepte & Implementierung, April 2018. (cited on Page and 6)

[Tic82] Walter F. Tichy. Design, implementation, and evaluation of a revision control system. In *Proceedings of the 6th International Conference on Software Engineering*, ICSE '82, page 58–67, Washington, DC, USA, 1982. IEEE Computer Society Press. (cited on Page 10)

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. pages 107–119, 1999. (cited on Page 102)

[Tor] Linus Torvalds. Linux Operating System. www.kernel.org. Accessed at December 02, 2019. (cited on Page 8)

[TSG+19] Jan-Philipp Tauscher, Fabian Wolf Schottky, Steve Grogorick, Paul Maximilian Bittner, Maryam Mustafa, and Marcus Magnor. Immersive EEG: Evaluating Electroencephalography in Virtual Reality. In *Proc. IEEE Virtual Reality (VR) Workshop*, pages 1794–1800, Mar 2019. PerGraVAR. (cited on Page , 20, and 21)

[vdZ] Erik van der Zalm. Marlin Firmware. http://marlinfw.org/. Accessed at December 02, 2019. (cited on Page 3, 7, 8, 74, 99, 106, and 111)

[vRGA+15] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. Presence-Condition Simplification in Highly Configurable Systems. pages 178–188, 2015. (cited on Page 62, 70, and 111)

[W+04] J Wiegand et al. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004. (cited on Page 102)

[wGM02] Szyperski, Clemens with Gruntz, Dominik and Murer, Stephan. Component Software – Beyond Object-Oriented Programming. Addison-Wesley, 2002. (cited on Page 102)

[WKP15] Jens H. Weber, Anita Katahoire, and Morgan Price. Uncovering Variability Models for Software Ecosystems from Multi-Repository Structures. pages 103:103–103:108, 2015. (cited on Page 57 and 103)

[WSSS16] David Wille, Sandro Schulze, Christoph Seidl, and Ina Schaefer. Custom-Tailored Variability Mining for Block-Based Languages. pages 271–282, March 2016. (cited on Page 1, 2, 11, and 103)

[XXJ12] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. Feature Location in a Collection of Product Variants. pages 145–154, 2012. (cited on Page 57 and 103)

[ZHP+14] Tewfik Ziadi, Christopher Henard, Mike Papadakis, Mikal Ziane, and Yves Le Traon. Towards a Language-Independent Approach for Reverse-Engineering of Software Product Lines. pages 1064–1071, 2014. (cited on Page 2 and 103)

https://doi.org/10.24355/dbbs.084-202002271120-0

https://doi.org/10.24355/dbbs.084-202002271120-0

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Braunschweig, den 29. Januar 2020

https://doi.org/10.24355/dbbs.084-202002271120-0