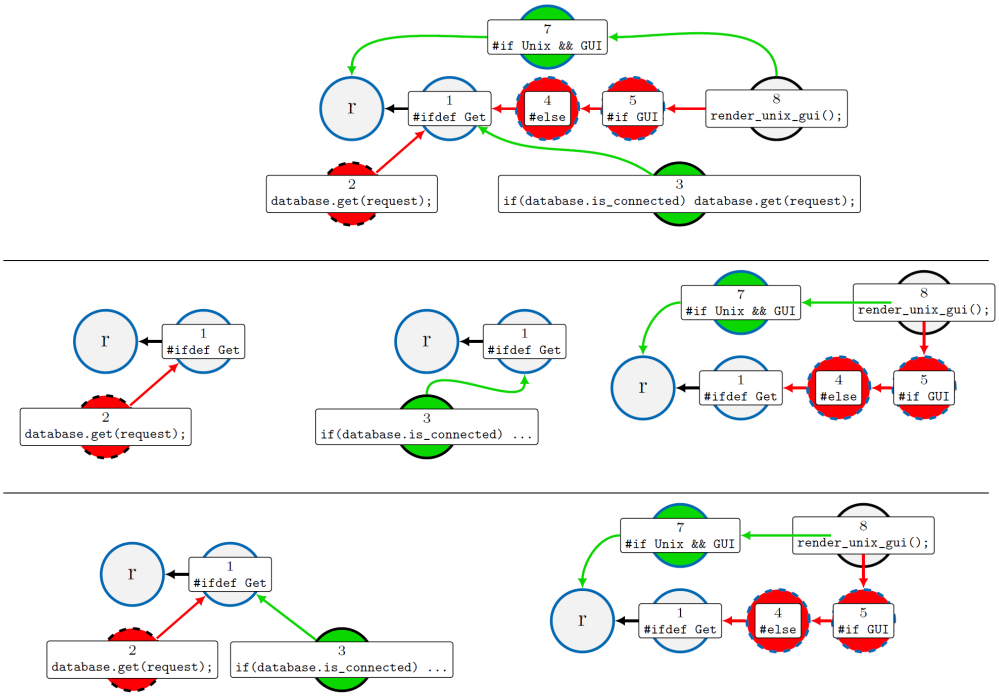
Software Engineering
Programming Languages

universität ulm



Reverse Engineering Feature-Aware Commits from Software Product-Line Repositories

Lukas Philipp Bormann

Bachelor's Thesis

Institute of Software Engineering and Programming Languages

Lukas Philipp Bormann:
Reverse Engineering Feature-Aware Commits from Software Product-Line Repositories

Advisors:

M.Sc. Paul Maximilian Bittner
Department of Software Engineering and Programming Languages

M.Sc. Christof Tinnes
Siemens AG, München

Prof. Dr.-Ing. Thomas Thüm
Department of Software Engineering and Programming Languages

Bachelor's Thesis, University of Ulm, 2022.

Abstract

Version control systems such as Git and Subversion are widely used. Even though software often has to allow for variability, the commonly used version control systems do not provide functionality to handle variability explicitly in a commit. This lack of functionality for software with high variability, like software product lines, results in time intensive development processes, reduced clarity of the commit history, and no possibility to propagate or revert individual changes to features. In this thesis, we introduce our operator FeatureSplit which addresses these problems, by reducing the complexity of the commit history of software product line repositories. FeatureSplit achieves complexity reduction through separation and regrouping of committed changes by feature. In an empirical evaluation, we show that the resulting *feature-aware commits* simplify the managing and referencing of variability in version control system. FeatureSplit simplified on average 88% of commits in the evaluated repositories.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

Version control systems such as Git and Subversion are widely used both in open-source projects and in the industry. A version control system is designed to track changes made to submitted content and group these changes chronologically with commits. A commit is a chronologically ordered wrapper of changes to the tracked content, that a developer submits to a repository to be logged. This allows developers to archive changes to tracked content and to store additional metadata of when, where, and who submitted the changes, with the changed content [4] [5] [6].

Even though software often has to allow for variability to be applicable in different environments, commonly used version control systems do not provide functionality to handle variability in a commit. Software variability describes the ability to change certain functions or features, before or in the compiling process of the software, to adapt to different market segments or contexts of use [7]. A feature is a domain abstraction that represents requirements, similarities, or differences of variability in software [8]. This can either be achieved by developing multiple versions of the same system simultaneously or by implementing variability into the source code itself. The problem of missing support for variability in version control systems can be illustrated with the following example: If a developer wants to revert a change to a specific feature, the developer has to revert the whole commit, which tracks the change. This means changes, which are not part of the feature but are tracked in that commit as well, will also be reverted. After such a revert the developer has to manually handle unwanted reverts, even though some changes might not affect the feature at all and are just part of the commit. This described problem would not happen if a commit would only contain changes to a specific feature. In an ideal scenario, version control systems support feature-aware commits when developers submit small and fine-grained commits, about a certain feature. Yet in practice, commits are rather large with many loosely related changes across features [4] [9]. A way to mitigate this problem would be to force the developers to only commit changes after the best practice of creating only single-task commits [4], but this is not realistically feasible [5].

Existing post-commit and commit-visualizing assistants, do not provide guidelines or restrictions to enforce the above-described best practice when working with variability

in version control systems. Many operators decompose composite commits, to improve readability [4] [6] [10]. For example, the *SmartCommit* operator splits commits into different developer activities, to improve the readability of commits [11], but SmartCommit and the other cited papers neglect variability in software systems. Other approaches control or visualize variability in software systems, but didn't apply them to commits in version control systems [12] [13] For example, *INFOX* identifies uncommitted features and regroups them into separate branches, which helps with the development process of features [14] but does not provide functionality to simplify a commit itself.

Thus, we propose the decomposition of commits into feature-aware commits, which can be achieved with our operator *FeatureSplit*. A feature-aware commit is a commit that only affects a single feature. This kind of commit is created if either a composite commit is reduced to multiple feature-aware commits or if the developer follows best practices and avoids composite commits altogether. FeatureSplit allows for extraction of feature-aware commits from a composite commit. FeatureSplit works in three steps, first, our operator takes a graph-based representation of a commit called *variation diff* [1] [15] and splits the commit into atomic diffs. An atomic diff is a variation diff that cannot be divided further without invalidating the variation diff. Second, the atomic diffs are clustered based on a desired or automatically generated feature, resulting in two clusters an atomic diff can be grouped to. The *feature-aware cluster* which holds all atomic diffs with the provided feature and the *remainder cluster* containing the remaining atomic diffs. Third, the clustered atomic diffs are composed into feature-aware variation diffs. The resulting two feature-aware variation diffs can be transformed to feature-aware commits, altered, or evaluated.

To quantitatively evaluate our proposed operator, we introduce the following research questions:

- **RQ1 (Accuracy):** To what extent is our operator accurate when creating feature-aware commits
- **RQ1 (Performance):** How performant is our operator and does it scale with larger input sizes

This thesis is outlined as follows. First, in Chapter 2, we explain the foundations of software product lines, version control systems, and the difference between committed changes commits and committed source code. In Chapter 3, we define the foundations our operator is based on. This lays the groundwork for Chapter 4, where we present our implementation and structure of our operator. We also introduce and answer our research questions there. After the implementation, in Chapter 5 we evaluate FeatureSplit based on a set of sample repositories. In Chapter 6 we place our thesis into the context of other related papers. In Chapter 7, we summarize the insights of this thesis and the capability of FeatureSplit. In the final chapter, Chapter 8, we provide an outlook for possible extensions and applications of FeatureSplit.

2. Background

In this chapter, we display background information necessary to understand this thesis. We will briefly explain the following concepts: Software product lines, version control systems, and the difference between committed changes, commits, and committed source code.

The feature-aware commits we create aim for better usability of software product lines in version control systems. To illustrate the interaction between software product lines and version control systems, we will use feature models to visualize the valid configurations of software product lines, presence conditions to apply the feature models to the implementation artifacts, and git diff notation to describe the changes in a commit.

In Section 2.1 we introduce fundamental comprehension of software product lines. In the subsequent Section 2.2, version control systems are defined.

2.1 Software Product Lines

A software product line is a set of program variants that share a common set of features. It is tailored to a common domain (market segment), with the goal of reusability of common software artifacts. Thus, it's enabling software evolution across multiple software variants without re-implementation of features or bug fixes in each variant. A feature is a domain abstraction that represents requirements, similarities, or differences of product variants [8]. Essentially, a product variant is a concrete combination of features. The disadvantage of software product lines is the required high initial coordination of domain engineering and the implementation of a generator. The generator takes the software configuration and the integrated codebase as input and returns a valid software variant. A central part of the development of software product lines is *feature modeling*, which captures the entirety of all valid configurations of the software product line on a conceptual level. They can be visualized with a *feature diagram* or mathematically expressed with propositional formulas.

In Figure 2.1, a feature diagram of a configurable database product line is visualized. The feature diagram is a tree representation, where each feature has a parent feature, except

for the root feature, and each feature groups zero more features, except for terminal features. Features can be selected based on the decomposition types [16] [17]:

- **Alternative Group:** Exactly one of the feature's sub-features must be included. An example can be seen in Figure 2.1 under the `OS` feature.
- **Or-Group:** At minimum one of the feature's sub-features must be included. In Figure 2.1 the feature `API` showcases the or-group.
- **And-Group:** All sub-features are included if a sub-feature is labeled *mandatory* or can be included if a sub-feature is *optional*. An example is the root feature of Figure 2.1.

Features can be labeled either *abstract* or *concrete*. An abstract feature does not have an implementation artifact, to which the feature is referring, concrete features reference specific implementation artifacts. An implementation artifact is a part of source code, which is identifiable and can have any range of granularity [1]. Additionally, specific cross-tree constraints can be specified which provide sub-tree overarching configurations. In Figure 2.1 the cross-tree constraint between `Win` and `GUI` is displayed below the feature diagram.

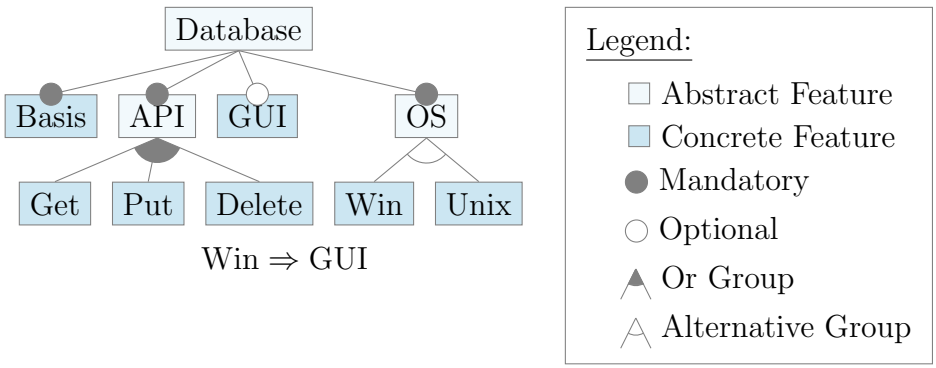


Figure 2.1: Example feature model representing a database product-line based on [2] [3].

To analyze the validity of the feature diagram and selected configuration of software variants, the feature diagram is translated into a propositional formula. Table 2.1 shows a possible translation from language constructs in feature diagrams to a propositional formula [3]. In Equation 2.1 an applied conversion of the feature diagram in Figure 2.1 is demonstrated. Equation 2.1a shows the root node, it always returns true and is only included for completeness. Equation 2.1b displays the optional sub-features of the `Database` and in combination with Equation 2.1c displays a simplification for the mandatory sub-features of the `Database`. Equation 2.1d shows the or-group of the `API` feature. Finally, in Equation 2.1e and f the alternative group is demonstrated with the `OS` feature.

$$\text{Database} \wedge \tag{2.1a}$$
$$(\text{Database} \Rightarrow \text{Basis} \wedge \text{API} \wedge \text{OS}) \wedge \tag{2.1b}$$
$$(\text{Basis} \vee \text{API} \vee \text{GUI} \vee \text{OS} \Rightarrow \text{Database}) \wedge \tag{2.1c}$$
$$(\text{API} \Leftrightarrow \text{Get} \vee \text{Put} \vee \text{Delete}) \wedge \tag{2.1d}$$
$$(\text{Win} \Leftrightarrow (\neg\text{Unix} \wedge \text{OS})) \wedge \tag{2.1e}$$
$$(\text{Unix} \Leftrightarrow (\neg\text{Win} \wedge \text{OS})) \wedge \tag{2.1f}$$
$$(\text{Win} \Rightarrow \text{GUI}) \tag{2.1g}$$

Feature Diagram	Propositional Formula	Example
Root R	R	Equation 2.1a
Optional C_i	$C_i \Rightarrow P$	Equation 2.1b
Mandatory C_i	$C_i \Leftrightarrow P$	Equation 2.1b, c
Or-Group (Inclusive-Or)	$P \Leftrightarrow C_1 \vee \ldots \vee C_n$	Equation 2.1d
Alternative Group (Exclusive-Or)	$(C_1 \Leftrightarrow (\neg C_2 \wedge ... \wedge \neg C_n \wedge P)) \wedge$ $(C_2 \Leftrightarrow (\neg C_1 \wedge ... \wedge \neg C_n \wedge P)) \wedge$ $\vdots$ $(C_n \Leftrightarrow (\neg C_1 \wedge \neg C_2 \wedge ... \wedge \neg C_{n-1} \wedge P))$	Equation 2.1e Equation 2.1f

Table 2.1: Translating a feature model to a propositional formula [2] [3].
R represents the root feature, P represents a parent feature, and C_i is the i-th child feature.

To compile feature configurations based on a feature model to a software variant, the generator of the software product line relies on *feature mappings*. A feature mapping is the process of referencing implementation artifacts to specific features or feature combinations. This can be achieved with different strategies (e.g. Frameworks, Services, or runtime variability). The strategy that is presented in this thesis is a conditional compilation with the C preprocessor, which is implemented in prominent projects like Marlin, the Linux kernel, and Vim [1]. The C preprocessor splits up source code depending on the preprocessor directives `#if`, `#ifdef`, and `#ifndef`, where `#ifdef` and `#ifndef` include or exclude implementation artifacts based on the presence of a certain macro and `#if` is utilized for complex conditions [16]. An example can be seen in Listing 2.1. Line `#ifdef Get` indicates, that if the feature `Get` is selected, line 2 will be included in the compiled source code. The `#else` statement in line 3 indicates, that if `Get` is not selected, lines 4-7 are compiled instead. The resulting code snippet, which is linked to a specific feature or feature combination is called a *feature trace*.

```
#ifdef Get
    database.get(request);
#else
#ifdef GUI
    render_unix_gui();
#endif
#endif
```

Listing 2.1: Example of the C preprocessor, based on Figure 2.1.

2.2 Version Control Systems

A version control system is a broad term used to describe software that allows for collaborative development of evolving objects, by allowing developers to work on the same elements simultaneously. A typical application of a version control system is the development of source code. In the following, we will focus on the usage of version control systems in software development. Version control systems accomplish simultaneous development, by managing changes that have been made to the source code, along with metadata about whom, when, and why changes have been made.

```
 #ifdef Get
-    database.get(request);
+    if (database.is_connected) database.get(request);
-#else
-#ifdef GUI
+#endif
+#if Unix && GUI
     render_unix_gui();
-#endif
 #endif
```

Listing 2.2: Example of the git diff notation, based on Listing 2.1.

Our operator has the potential to be applied to any version control system, in this thesis the development and evaluation of our operator are based on the control system Git. If a developer wants to integrate a version control system into a project, he has to initialize a *repository*. The repository contains the whole project evolution and allows reviewing of older versions of the source code. In Git, the repository is decentralized and every local copy of the repository can replace the main repository at any time. A repository by itself does not contain any information about the project's evolution. To track the software evolution a developer has to send a *commit* to store the current progress of the development inside the repository. A commit can be defined as a wrapper for chronologically related changes to a source code, it stores who, when, and why the changes in the commit were made. A commit can also function as a categorical wrapper, but this decision is up to the developer and is currently not enforced in any way in version control systems. In the repository, a commit acts as a reference point to find *committed source code*. The committed source code in state-oriented version control systems like git consists of tree structure and blob files, which compresses the contents of the current working directory. If the source code changes a new blob file is created. We define the difference between two commits or states of the repository as *committed changes* [18] [19]. Com-

mitted changes can be visualized with the *git-diff notation*. A git-diff tracks line-based changes. In the git-diff notation, added lines are proceeded by a '+'. Removed lines are proceeded by a '−'. If a line is altered, the original line is considered to be removed and the altered is considered to be added. The git-diff notation is visualized in Listing 2.2. For example, lines 2, 4, 5, and 9 are removed, lines 3, 6, and 7 are added, and lines 1, 8, and 10 did not change.

3. Reverse Engineering Feature-Aware Commits

In this chapter, we introduce the core concept of this thesis, the extraction of feature-aware commits from commits. The separation of commits into feature-aware commits has different applications. Developers in software product lines can, for example, create more structured commits before creating a push-request or developers can gain more precise insight into the commit history of a feature, when only feature-aware commits in the commit history are displayed, that affect the given feature.

First, we have to find a model to describe edits to feature mappings and corresponding implementation artifacts. This model is essential to allow our operator to analyze and manipulate commits containing features. Our operator utilizes this model for feature extraction and feature separation of commits. We use a tree-based representation of commits, as common in the literature [15], [11], [20], [14]. Second, we introduce our operator FeatureSplit. Our operator implements a three-step process, where a commit is first separated into sub-commits. The generated sub-commits are then regrouped depending on a given feature or feature formula that should be extracted. Finally, each group of sub-commits is composed into a feature-aware commit. Depending on the model that represents a commit, the splitting of a commit into sub-commits can have restrictions. In case of the tree-based model we integrated, to grant the validity of the model a sub-commit has to contain all ancestors of any leaf node present in the model. Valid separation of commits is discussed in detail in Section 3.3.1. We define resulting sub-commits as atomic, which means that the sub-commits can not be divided further without destroying validity of the tree-based model. Lastly in Section 3.3.3, we describe the creation of feature-aware commits, by composing multiple atomic commits which affect a certain feature. An atomic commit can also be a feature-aware commit if it is the only atomic commit affecting a certain feature.

To represent commits, our operator uses variation diffs introduced by Bittner et al. [1]. We describe variation diffs in Section 3.2 and variation trees, which are the foundation for variation diffs in Section 3.1. In Section 3.3, we describe the concept of how our operator separates variation diffs correctly into atomic variation diffs which we define

as *atomic diffs*. Additionally, in Section 3.3 we show our approach for combining atomic diffs into feature-aware variation diffs.

3.1 Variation Trees [1]

In this section, we explain variation trees described by Bittner et al. [1]. Our operator requires a model which describes edits to feature mappings and corresponding implementation artifacts. The consideration of feature mappings and feature mapping edits is required to identify and isolate a feature and its corresponding implementation artifacts. It allows us to cluster and combine changes to the same feature or feature formula, creating feature-aware commits. To narrow our decision between different models that represent commits, we based our decision on following prerequisites. First, we need a model that identifies which implementation artifacts were added, removed, or edited. Identification of changed implementation artifacts allows us to solely interact with and gather information about changes without identifying changed implementation artifacts ourselves. Second, the model should also cover how the changes affect the variability in the source code, which is required for the clustering process of implementation artifacts by feature. *Variation tree* and *variation diffs* by Bittner et al. [1] were designed to meet these requirements. Variation diffs are used in FeatureSplit to describe edits to feature mappings and source code.

Fundamentally, a *variation tree* is a formalization, that constitutes nesting in feature mappings. It is a tree representation of feature mappings applied to implementation artifacts, such as source code. The distinction between feature mappings and implementation artifacts in the tree structure allows us to differentiate between feature mapping changes and implementation artifact changes, which is necessary to cluster implementation artifacts correctly.

A variation tree (V, E, r, τ, l) consists of the following elements [1]:

- A set of nodes V: A tree node $v \in V$ represents either an implementation artifact or a feature mapping. It has a type $\tau(v) \in \{\texttt{artifact}, \texttt{mapping}, \texttt{else}\}$, which describes the content of the node's label $l(v)$, which holds an implementation artifact or feature mapping, respectively.
- A set of edges $E \subseteq V \times V$: A directed edge $e \in E$ describes a hierarchical relationship between two nodes. A child node pointing a directed edge to a parent node symbolizes that the child node is part of the parent node. The parent node y of a child node x is referenced by, with the expression $p(x) = y$. Each node can have an arbitrary, but finite number of children. The root node r does not have parents and all other nodes have at least one parent.
- $r \in V$: is the root node of the variation tree. It is not part of the source code and represents a neutral feature mapping. It is essential for grouping all artifacts and mappings. The root node has the type $\tau(v) = \texttt{mapping}$ and the label $l(v) = \text{true}$.
- $\tau : V \to \{\texttt{artifact}, \texttt{mapping}, \texttt{else}\}$: The node type $\tau(v)$ describes if a tree node v contains source code or a feature mapping. The type `artifact` indicates, that a node represents any kind of implementation artifact, a node with the `mapping` type

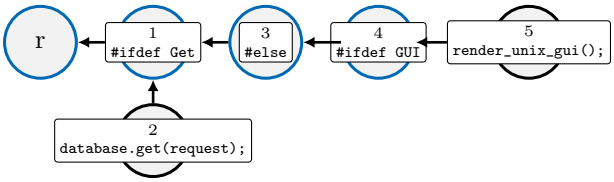


Figure 3.1: Variation tree of Listing 2.1

contains a propositional formula, and a node with the `else` type can only exist if the parent node of the `else` node is a `mapping` node. An `else` node represents the negation of its parent `mapping` node's feature mapping.

- $l(v)$: The label $l(v)$ of a node contains different information based on the node type. An `artifact` node references the implementation artifact in the label. A `mapping` type node contains a propositional formula in the label and for an `else` node the label is empty.

As an example, Figure 3.1 shows a visual representation of the variation tree of the code in Listing 2.1. A node with a blue border visualizes an annotation, which are nodes of type $\tau(v) = \texttt{mapping}$. An example of an annotation is the `#if Get` node. A node with a black border represents an implementation artifact, such as the `render_unix_gui()` node. Finally, at the top of a variation tree is the root node, which is used for grouping nodes.

Considering that a variation tree represents the nesting hierarchy of feature mappings, we can compute valid feature mappings of an artifact given its corresponding node n, with the function shown in Equation 3.1 [1]. To extract the feature mapping of a node, the variation tree has to be traversed upwards until a node of type `mapping` is found. First, for each node with the type `artifact`, the parent node is inspected. Second, for each `mapping` node, its feature mapping is extracted from its label. Finally, for each node with type `else` the label of its parent node is negated. To illustrate Equation 3.1, we are applying this function to different nodes in Figure 3.1. Node 2 is an `artifact` node. The variation tree is followed upwards until the next `mapping` node is found, which is node `1`. The label of node `1` is returned, resulting in the feature mapping `Get`. Node `3` is an `else` node. Equation 3.1 returns for this node the negated label of the parent node `1`, which is `¬Get`. Node `4` is already a `mapping` node and its label is returned, resulting in a feature mapping of `GUI`.

$$\mathrm{F}(n) := \begin{cases} \mathrm{F}(p(n)), & \tau(n) = \text{artifact}, \\ l(n), & \tau(n) = \text{mapping}, \\ \neg\mathrm{F}(p(n)), & \tau(n) = \text{else}. \end{cases} \tag{3.1}$$

A variation tree also allows for the computation of a node's *presence condition*. A presence condition is a propositional formula that defines when an implementation artifact is included in a variant [21]. If a presence condition evaluates to `true` for a certain feature configuration, the corresponding implementation artifacts are included in the software variant. A presence condition is the combination of all feature mappings that affect the implementation artifact. A function to compute the presence condition is formulated

in Equation 3.2 [1]. To compute a presence condition, every ancestor above the given node is recursively inspected, until arriving at the root node. First, `artifact` nodes are skipped, because they do not affect the presence condition of the initial node. Second, the feature mappings of `mapping` nodes are conjunct $\wedge$ with feature mappings of previously visited `mapping` nodes. Third, the feature mapping of an `else` node is conjunct with the presence condition of its corresponding `mapping` node's parent $p(p(n))$. Fourth, if the root is reached the traversal ends. The root node acts neutral in conjunction, due to its definition $F(r) = l(r) = \text{true}$. As an example, the presence condition of node 5 in Figure 3.1 is generated in three recursive steps. First, node 5 is skipped, because it is an `artifact` node. Then, the label of the parent node 4 is added, because it is a `mapping` node. Node 3 is an `else` node, resulting in the negation of its parent node, node 1. Lastly, the label of the root node is added to the presence condition, resulting in the presence condition: $\texttt{GUI} \wedge \neg \texttt{Get} \wedge true$.

$$\text{PC}(n) := \begin{cases} \text{PC}(p(n)), & \tau(n) = \text{artifact}, \\ F(n) \wedge \text{PC}(p(n)), & \tau(n) = \text{mapping}, n \neq r, \\ F(n) \wedge \text{PC}(p(p(n))), & \tau(n) = \text{else}, \\ F(n), & n = r. \end{cases} \tag{3.2}$$

A variation tree is considered to be complete and sound [1]. Soundness and completeness together mean that we can derive all and only valid arguments of a system [22]. Concerning variation trees, completeness means that any edit in a commit can be modelled by a valid variation tree. Soundness describes, that every variation tree represents a possible edit in a commit. Completeness arises due to the fact, that `mapping` nodes are general enough to support any kind of conditional annotation in form of a propositional formula, and `artifact` nodes can contain any kind of implementation artifact. Soundness is argued to be given in variation trees because in every possible variation tree, the feature mapping and the PC of a node are defined and thus variability is always specified.

3.2 Variation Diffs [1]

In this section, we explain variation diffs described in [1]. A variation diff represents the difference between two variation trees. In this thesis, we utilize variation diffs to describe the differences introduced by a patch in a commit. A patch represents one or multiple edits in a certain file. A set of patches committed at the same time represent a commit. Variation diffs allow us to describe and analyze committed changes to implementation artifacts and feature mappings. With variation diffs, we can describe changes to variability in a commit.

A variation diff $(V, E, r, \tau, l, \Delta)$ is a rooted directed connected acyclic graph, which extends a variation tree [1]. A variation diff combines two variation trees, by additionally storing a function $\Delta : V \cup E \to \{+, -, \bullet\}$ that indicates if a node or edge was added $+$, removed $-$, or stayed unchanged $\bullet$.

An example of a variation diff is shown in Figure 3.2. This graph shows changes to feature mappings (nodes 4, 5, and 7) and edits of implementation artifacts (nodes 2 and 3).

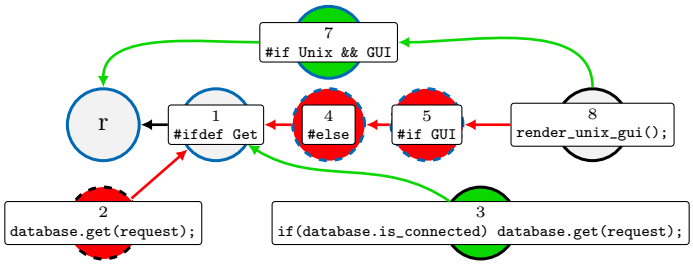


Figure 3.2: Variation diff of the diff in Listing 2.2

Green nodes in the variation diff are nodes that are inserted into the variation tree. Node 3 is an example of an added implementation artifact. Red nodes with dotted outlines are removed from the variation tree. For example, node 2 is a removed node. Finally, gray nodes are unaltered and identical in both variation trees. An example of an unchanged node is node 8.

The variation tree before or after a variation diff is obtained by a *projection* of the variation diff [1]. The projection function is formulated in Equation 3.3 [1]. The projection uses time $t \in \{b, a\}$ to determine whether a variation tree that is obtained should resemble the variation tree before or after the edit. The function returns every node and edge that *exists* at time t. To determine if a node or edge exists at a given time t, Equation 3.4 is defined [1]. A node or edge $x \in V \cup E$ *exists* before (b) the edit if the node or edge was not added to the variation tree, and *exists* after (a) the edit if the node was not deleted.

$$\begin{aligned} \text{project}((V, E, r, \tau, l, \Delta), t) := &(\{v \in V | \text{exists}(t, \Delta(v))\}, \\ &\{e \in E | \text{exists}(t, \Delta(e))\}, \\ &r, \tau, l) \end{aligned} \tag{3.3}$$

An example of a before projection from the variation diff displayed in Figure 3.2, can be seen in Figure 3.1. We can utilize the projection to inspect the resulting implementation artifacts and feature mappings after applying only certain feature-aware commits generated from the initial commit.

$$\text{exists}(t, d) := (t = b \wedge d \neq +) \vee (t = a \wedge d \neq -) \tag{3.4}$$

Variation diffs are proven to be complete and sound regarding edits to variation trees [1]. Variation diffs can describe any edit to a variation tree. This is proven to be true: Any edit to a variation tree can be described by first removing the whole *before* variation tree and then adding the whole *after* variation tree again. Variation diffs describe an edit between two committed states of a software product line. This is proven to be true because a variation diff always describes a valid edit. The proof is that every variation diff by definition always describes a state before the edit and a state after the edit via the project function described in Equation 3.3. For this reason a variation diff always describes a change between two committed states.

3.3 Extracting Feature-Aware Commits

The contribution of this thesis is a transformation of a composite commit into one or multiple feature-aware commits. A composite commit describes a commit that contains changes affecting multiple features. We define a feature-aware commit as a commit that contains changes to exactly one feature or exactly one feature formula.

To transform a composite commit into one or multiple feature-aware commits, we manipulate the variation diffs of patches in a composite commit. We achieve this manipulation with our operator FeatureSplit which is visualized in Figure 3.3. Our operator incorporates variation diffs, by accepting a variation diff together with a feature query as input. A feature query is a propositional formula describing which feature or feature combination should be extracted from the initial commit. The input is then manipulated in a three-step process, visualized as rounded nodes in the middle column of Figure 3.3. Square nodes represent data objects used and generated in FeatureSplit. First, our operator separates variation diffs into multiple atomic diffs, which are variation diffs that cannot be divided further without invalidating the soundness of the extracted atomic diffs. Second, extracted atomic diffs are clustered in either the feature query cluster or the remainder cluster depending on the feature or feature formula in the feature query. If an atomic diff affects a feature described in a given feature query the atomic diff is added to the feature query cluster, otherwise, the atomic diff is added to the remainder cluster. Third, clustered atomic diffs are composed into single feature-aware variation diffs, where the variation diff of the feature query cluster only contains edits to the feature or feature combination asked for in the feature query. The remainder cluster is composed into a variation diff which does not contain any feature or feature combination asked for in the feature query.

In Section 3.3.1, we describe the first step of FeatureSplit. The first step identifies all edits in a variation diff and extracts atomic diffs based on the identified edits. In Section 3.3.2, we describe the second step, showcasing how atomic diffs are clustered together into feature-aware clusters. A given feature query is used to determine which atomic diffs are sorted into which cluster. In Section 3.3.3, we describe the third step, displaying how each cluster is composed into a single feature-aware variation diff. The process of transforming a commit into feature-aware commits applies to a single commit. In Section 3.4, we display a process to extract relevant features from a variation diff which can be converted to feature queries and used for automatic feature extraction from commits. Lastly, in Section 3.5 we provide an example of each process in FeatureSplit.

3.3.1 Step 1: Variation Diff Separation

In the first phase, we split a composite commit in form of a variation diff into multiple atomic diffs. To achieve this, we present Algorithm 1. The algorithm extracts a set of atomic diffs which as a whole resemble the initial variation diff. The initial variation diff is passed to the algorithm as a parameter, which can be seen in Line 1. In Line 2 we initialize a temporary set to store the extracted atomic diffs in. The temporary set removes duplications of atomic diffs, which occur if at least one ancestor of an edited node was edited as well. In lines 3 and 4, we filter all nodes that were edited. The filtering is necessary to find all is atomic diffs. We also could extract a atomic diff for every node in a variation diff, but only inspecting the edited nodes is sufficient, because

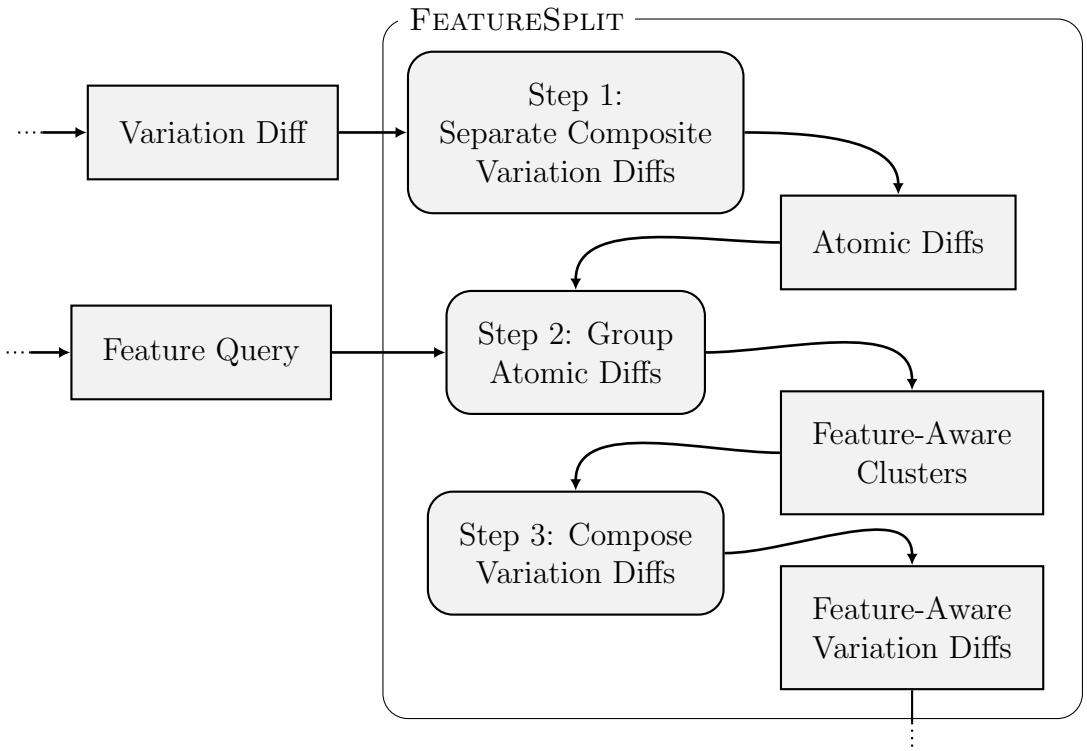


Figure 3.3: Pipeline of FeatureSplit

unedited nodes only exists in a variation diff, if the unedited node is a descendant or ancestor of an edited node. In Line 5 we extract for each edited node its atomic diff. The extraction of an atomic diff is described in Algorithm 2. For efficiency and simplicity the atomic diff extraction algorithm returns only nodes and edges. The extraction of only nodes and edges is sufficient because we can extract valid atomic diff based on the returned nodes and edges from Algorithm 2 in combination with the existing root node and functions provided by the initial variation diff. Only returning nodes and edges is possible, because the extraction does not alter the root node or the functions τ, l and Γ provided in variation diff. In Line 6 we use the returned nodes and edges to create a new variation diff which represents an atomic diff. The resulting atomic diff is then added to the temporary set of other atomic diffs. Lastly, in Line 9 the set of atomic diffs is returned.

The extraction of atomic diffs utilized in Algorithm 1 is described in Algorithm 2. Algorithm 2 extracts, based on a given node v, an atomic diff from a given variation diff, by extracting a minimal and valid variation diff that includes node v. Generally, an atomic diff is extracted by inspecting the ancestors of a node v and the ancestors of the descendants of v, and extracting these inspected nodes and edges into the new atomic diff. In Line 2 of Algorithm 2, the parameters for the algorithm are defined. The algorithm requires a node v and a variation diff D. The node v is used to extract from D an atomic diff that includes v. In Line 3, we initialize temporary sets for nodes and edges containing nodes and edges which will be converted into an atomic diff. In Line 4, we utilize Equation 3.5 to extract all child nodes n from v. Equation 3.5 identifies all edges where node v is the parent node and returns the child node of each edge:

Algorithm 1: Extraction of all valid atomic diffs

Input : $D := (V, E, r, \tau, l, \Delta)$, a variation diff of a committed change.
Output: A set $\{d_i\}, i = 1, 2, ..., n$, where each element d_i is an atomic diff of D. An atomic diff d_i is a valid variation diff

```
1  Function separateDiff(D = (V, E, r, τ, l, Δ)):
2      solutionSet ← ∅
3      forall v ∈ V do
4          if Δ(v) ≠ • then
5              V_v, E_v ← extractAtomicDiff(v, D)
6              solutionSet ← solutionSet ∪ {(V_v, E_v, r, τ, l, Δ)}
7          end if
8      end forall
9      return solutionSet
10
```

$$\text{children}(v, (V, E, r, \tau, l, \Delta)) := \{c | (c, v) \in E\} \tag{3.5}$$

In Line 5, we use Equation 3.6 to find all ancestor nodes of a given node n and add every ancestor node to the set of nodes V_n which describes the atomic diff. To find all ancestor nodes, Equation 3.6 requires a start node n from which the ancestors are extracted. The root node r of the initial variation diff, is used as the breakpoint for the recursive extraction of ancestors. Lastly, the time t is required, which allows us to define if the ancestor before or after the commit should be extracted. To extract all ancestors, Equation 3.6 has to be called twice as described in Line 5, once for all ancestor nodes before and once for all ancestor nodes after the commit.

$$\text{ancestorNodes}(n, r, t) = \begin{cases} \{n\} \cup \text{ancestorNodes}(p_t(n), r, t), & n \neq r \\ \{n\}, & n = r \end{cases} \tag{3.6}$$

Similar to lines 5 and 6, we use Equation 3.7 to find all edges between the traversed ancestor nodes of a given node n and add every traversed edge to the set of edges E_n. Equation 3.7 works similar to the earlier defined Equation 3.6 with difference in the returned values. Equation 3.7 returns the edge between a node and its parent at time t. As in Equation 3.6, to find all ancestor edges Equation 3.7 has to be called twice, once for all edges before and once for all edges after the commit.

$$\text{ancestorEdges}(n, r, t) = \begin{cases} \{(n, p_t(n))\} \cup \text{ancestorEdges}(p_t(n), r, t), & n \neq r \\ \emptyset, & n = r \end{cases} \tag{3.7}$$

In lines 7 to 12, Algorithm 2 is recursively called for every node that is not a leaf node. From Line 9 to Line 11, nodes and edges extracted from the recursion are joined to create a complete set of nodes and edges, which represent the atomic diff of the given node v. Finally, in Line 14, all nodes and edges from the atomic diff are returned. As we described

earlier, the returned nodes and edges are sufficient to create a complete atomic diff later on.

The extracted atomic diffs from the above-discussed variation diff separation process, are variation diffs designed to have a minimal amount of nodes without invalidating the definition of variation diffs. To guarantee the validity of the extracted variation diffs, every node except for the root node is required to have two parent nodes, one before parent and one after parent. If this criterion is not met, the projection of a variation diff with nodes that have only one parent node would result in an invalid variation tree structure [1]. This means for the extraction process, that if we decide to include a node, we have to check if both parent nodes are already included and add them if not. To guarantee that each parent of an extracted node is included we recursively call Algorithm 2 to inspect if every added child node has both parent nodes defined.

Algorithm 2: Atomic diff extraction

Input : v, an edited node
Output: A set of edges E_n and nodes V_n which resemble the extracted atomic diff

```
1  /* Generate the complete atomic diff, linked to the node v        */
2  Function extractAtomicDiff(v, D = (V, E, r, τ, l, Δ)):
3      V_n, E_n ← ∅
4      forall n ∈ children(v, D) do
5          V_n ← V_n ∪ {findAncestorNodes(n, r, b)} ∪
               {findAncestorNodes(n, r, a)}
6          E_n ← E_n ∪ {findAncestorEdges(n, r, b)} ∪
               {findAncestorEdges(n, r, a)}
7          if children(n, D) ≠ ∅      /* Inspect grandchildren          */
8           then
9              V_m, E_m ← extractAtomicDiff(n, D)
10             V_n ← V_n ∪ V_m
11             E_n ← E_n ∪ E_m
12         end if
13     end forall
14     return V_n, E_n
15
```

3.3.2 Step 2: Variation Diff Clustering

In this phase, a set of variation diffs, which also can be a set of atomic diffs, is grouped into different feature-aware clusters. The grouping of variation diffs is achieved by linking a specific feature query to a specific cluster.

A feature query is a feature formula, which describes which feature or what feature formula should be included in a feature-aware commit. To identify if a variation diff should be included in a specific cluster, a boolean satisfiability problem has to be solved. The boolean satisfiability problem uses a feature query and the presence condition of any `artifact` node in a given variation diff, to determine if the node should be included. For

example, if the boolean satisfiability problem is applied to every node in Figure 3.2, with the desired feature query `GUI`, the variation diffs containing node `5` would be added to the `GUI` cluster, because `GUI` is present in the *before* presence condition: GUI $\wedge \neg$Get $\wedge\, true$. We discuss the boolean satisfiability problem and SAT solving in detail in Section 3.3.2.1.

To achieve the clustering, we define Algorithm 3. This algorithm takes a set of variation diffs D and a feature query Θ as input. For every variation diff, the propositional formula in the feature query is evaluated for a possible match, with Equation 3.8. If the evaluation of a propositional formula returns `true`, the variation diff is added to the corresponding feature-aware cluster and will not be added to the remainder cluster. The remainder cluster contains variation diffs with no connection to the feature or feature formula in the given feature query. This distinction provides the opportunity to extract single feature-aware commits from composite commits while leaving the remainder of the composite commit as is. Finally, the clustering algorithm then returns a list of feature-aware clusters, which are represented as sets of variation diffs.

Algorithm 3: Division of variation diffs into Cluster of variation diffs

Input : A set of variation diffs D that are clustered depending on the feature query Θ

Output: A feature-aware clusters containing with variation diffs affecting the feature query and a remainder cluster with the remaining variation diffs

```
1  Function cluster(D, Θ):
2      featureCluster ← ∅
3      remainderCluster ← ∅
4      foreach d ∈ D do
5          if evaluate (d, Θ) then
6              featureCluster ← featureCluster ∪ {d}
7          else
8              remainderCluster ← remainderCluster ∪ {d}
9          end if
10     end foreach
11     return (featureCluster, remainderCluster)
12
```

The above-mentioned `evaluate` function referenced in Algorithm 3, defines if a variation diff is added to a feature aware cluster or the remainder cluster. The `evaluate` function is formally described in Equation 3.8 and inspects every node in the variation diff, by calling `evaluateNode`, which is described in Equation 3.9. The `evaluateNode` function returns `true`, if the inspected node is a `mapping` node and `evaluateQuery` returns `true`. The evaluation of feature queries is further discussed in Section 3.3.2.1 with `evaluateQuery` described in Equation 3.10. In the `evaluateNode` function we are only inspecting `mapping` nodes, because `mapping` nodes contains all information regarding features and feature formulas. The result of the `evaluateNode` function is then used to determine if a variation diff has at least one connection to the given feature query. If a connection could be detected, the `evaluate` function also returns `true`, indicating that the current variation diff should be added to the cluster referenced to the feature query.

$$\text{evaluate}((V, E, r, \tau, l, \Delta), \Theta) := \bigvee_{v \in V} \text{evaluateNode}(v, \Theta) \tag{3.8}$$

$$\text{evaluateNode}(v, \Theta) := \text{evaluateQuery}(v, \Theta) \wedge \tau(v) = mapping \tag{3.9}$$

3.3.2.1 Evaluating Feature Queries

As described in the previous section, FeatureSplit uses a propositional formula to evaluate if a variation diff should be included in a certain cluster. FeatureSplit uses the `evaluateQuery` function described in Equation 3.10 to compare the feature query Θ with the presence condition of the provided node. Other possible propositional formula solvers could be implemented to extend the customizability of FeatureSplit, which are shortly discussed in Section 8.2. `EvaluateQuery` performs a tautology check to decide if the feature query is contained in the presence condition of the node. If all features described in the feature query are defined in the presence condition, it means the boolean satisfiability problem is solved. Otherwise, the propositional formula will return `false`.

$$\text{evaluateQuery}(v, \Theta) := PC(v) \models \Theta \tag{3.10}$$

3.3.3 Step 3: Variation Diff Composition

In this phase, a cluster of feature-aware variation diffs is composed into a feature-aware variation diff based on the feature-aware clusters generated in step 2. The composition of atomic diffs into feature-aware variation diffs is described in Equation 3.11. In this equation a pair of variation diffs (d_1, d_2) is taken from the cluster $\{d_1, ..., d_n\}$ and combined into one composite variation diff with Equation 3.12. After combining the two variation diffs, the resulting composite variation diff is added to the set of variation diffs from the cluster, that were not composed. Then Equation 3.11 is recursively called, until only two variation diffs remain, which are combined into the feature-aware variation diff representing the feature-aware cluster. If only one atomic diff is passed to the composition function, the atomic diff is just returned. Lastly, if the composition function is called with an empty set, an empty set is also returned.

$$\text{composition}(\{d_1, ..., d_n\}) := \begin{cases} \text{composition}(\Omega(d_1, d_2) \cup \{d_3, ..., d_n\}), & n > 2 \\ \Omega(d_1, d_2), & n = 2 \\ \{d_1\}, & n = 1 \\ \emptyset, & n = 0 \end{cases} \tag{3.11}$$

The Equation 3.12 composes two variation diffs by creating a union of all nodes and edges from the two variation diffs. The root nodes, the type function τ, and the label function l are not changed, because we assume that they do not differ between atomic diffs in a given cluster, due to the identical initial variation diff, the atomic diffs were generated from.

$$\Omega((V_1, E_1, r_1, \tau_1, l_1, \Delta_1), (V_2, E_2, r_2, \tau_2, l_2, \Delta_2)) := (V_1 \cup V_2, E_1 \cup E_2, r_1, \tau_1, l_1, \Delta_1) \tag{3.12}$$

We argue, that the above-described equations are sufficient to generate correct and valid composite variation diffs based on the atomic diffs in the provided cluster. We support our statement through the following conclusion. All atomic diffs generated from the initial variation diff represent a part or the whole of the initial variation diff. In the atomic diff extraction process, nodes and the relations between nodes are not altered, only duplications of nodes and edges are performed to provide a valid variation diff structure. Based on that statement, the union of two atomic diffs solely removes previously created duplications and recombines atomic diffs into a composite variation diff. Finally, because the union of nodes and edges also does not alter a node or an edge, we believe that FeatureSplit never alters the committed changes in the initial commit, FeatureSplit merely alters in which commit the committed changes appear.

3.4 Feature Query Generator

FeatureSplit requires a feature query to determine which atomic diff should be extracted from a variation diff. Depending on the use case, it can be desirable to either define a custom feature query or to automatically generate a feature query. In this section we discuss a concept of a feature query generator described in Equation 3.13. `FeatureQueryGen` described in Equation 3.13, takes a variation diff as input and returns a set of features described in the given variation diff.

$$\text{featureQueryGen}((V, E, r, \tau, l, \Delta)) := \bigcup_{v \in V} \text{featureMappingExtraction}(v, \tau, l) \tag{3.13}$$

To generate the set of features, `featureQueryGen` joins the extracted sets of features from `featureMappingExtraction` described in Equation 3.14.
`FeatureMappingExtraction` extracts features by inspecting the feature mapping of `mapping` nodes and passing the label to Definition `extractFeature` described in Definition 3.1.

$$\text{featureMappingExtraction}(v, \tau, l) := \begin{cases} \text{extractFeature}(l(v)), & \tau(v) = \text{mapping} \\ \emptyset, & \text{else} \end{cases} \tag{3.14}$$

`ExtractFeature` extracts the set of features from a propositional formula by defining rules defined from Equation 3.15b to Equation 3.15f. Equation 3.15b defines that a variable, which can be a feature, is returned as a set with the variable as element in it. The set is then joined with other sets which contain features, to create the set of features that represent the provided variation diff. Equation 3.15c removes the negation operator, to access the variable contained in the negation operator. Equation 3.15d and Equation 3.15e join multiple extracted feature sets with each other. Lastly, Equation 3.15e removes brackets which are redundant in a set of isolated features.

Definition 3.1 (extractFeature)**.** *Extracts a set of features from a given propositional formula. Let F be the set of all propositional formulas and V the set of possible variables.*

$$extractFeature : F \to P(V) \tag{3.15a}$$
$$extractFeature(v) = \{v\},\ \textit{if } v \textit{ is a variable} \tag{3.15b}$$
$$extractFeature(\neg f) = extractFeature(f) \tag{3.15c}$$
$$extractFeature(f \wedge g) = extractFeature(f) \cup extractFeature(g) \tag{3.15d}$$
$$extractFeature(f \vee g) = extractFeature(f) \cup extractFeature(g) \tag{3.15e}$$
$$extractFeature((f)) = extractFeature(f) \tag{3.15f}$$

3.5 Example

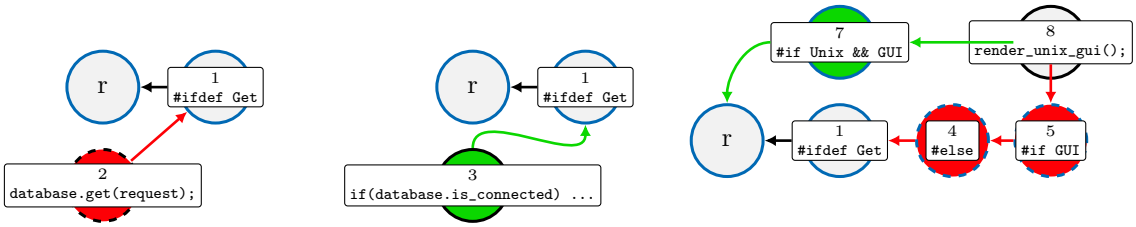
To fully illustrate the concept of our operator FeatureSplit we exemplify the process by using the diff in Listing 2.2. In previous steps, we already showed the transformation of the commit in Listing 2.2 into a variation diff, which can be seen in Figure 3.2.

The first step of FeatureSplit is the feature separation step. In this step, we split the variation diff in the smallest possible and still valid variation diffs, also called atomic diffs. Applied to the described example, we generate three atomic diffs which can be seen in Figure 3.4. The changes in atomic diff 3 cannot be separated further, without invalidating the variation diff. In more detail, the paths with nodes $\{8, 5, 4, 1, r\}$ and $\{8, 7, r\}$ cannot be extracted into atomic diffs due to Node 8, which is linked to Node 7 and Node 5. Splitting these paths into separated variation diffs would result in invalid variation diffs, as mentioned in Section 3.3.1

In the second step, the generated atomic diffs are grouped into feature-aware clusters. In this example, we define the feature query to cluster the feature-aware variation diffs as follows: `Unix`. We extracted the feature in the above defined feature query with the feature query generator mentioned in Section 3.4, by selecting the `Unix` feature from the received feature query set $\{\text{Get}, \text{Unix}, \text{GUI}\}$. The defined feature query is compared to ever `mapping` node to find if the presence condition includes the `Unix` feature mapping. The feature query `Unix` can only be found in only atomic diff 3. For this reason, the cluster `Unix` only contains a single variation diff. The remaining atomic diffs are grouped in the `remainder` cluster, which are atomic diffs 1 and 2.

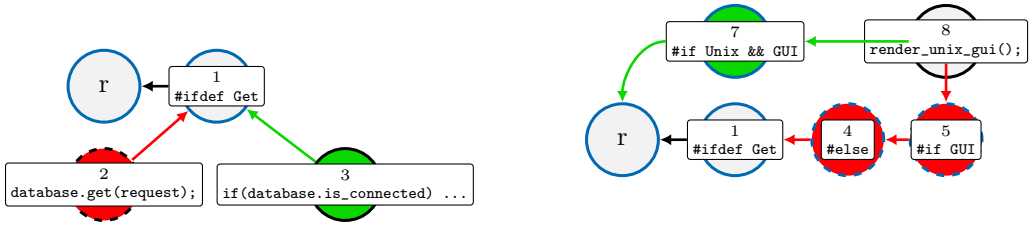
In the third and last step, the clusters are composed together to feature-aware variation diffs. As discussed earlier, because FeatureSplit does not alter the nodes or edges themselves, the atomic diffs can be composed by adding the missing nodes and edges from atomic diff to the other atomic diff. The atomic diff containing all nodes is then returned as a feature-aware variation diff. The `remainder` cluster is composed into the variation diff 3.5(a), by adding Node 3 from atomic diff 2 to Node 1 in atomic diff 1. The resulting atomic diff is returned as the `remainder` variation diff. Cluster `Unix` only contains one atomic diff, for this reason the atomic diff already represents the feature-aware variation diff. The resulting feature-aware variation diffs can be seen in Figure 3.5.

To extract multiple features from a variation diff FeatureSplit could be called sequentially, by using the `remainder` variation diff of the previous result as input for FeatureSplit with a new feature query.



(a) atomic diff 1 (b) atomic diff 2 (c) atomic diff 3

Figure 3.4: Subtrees of variation diff described in Figure 3.2



(a) Remainder variation diff (b) feature-aware variation diff

Figure 3.5: Feature-aware variation diffs based of the atomic diffs in Figure 3.4

3.6 Summary

In this chapter, we described the main concept of this thesis, which is the transformation of commits into feature-aware commits. We achieved this by designing two operators FeatureSplit and a feature query generator. To manipulate commits we described and integrated variation diffs into our operators. Variation diffs and variation trees are commit representation models introduced by Bittner et al.. Our first operator FeatureSplit, converts a patch of a commit represented as a variation diff into feature-aware variation diffs. Our second operator, a feature query generator, extracts every feature from a given variation diff. The implementation of both operators is thoroughly discussed in Chapter 4 and evaluated in Chapter 5.

4. Tool Support

In this chapter, we present an implementation of our operator FeatureSplit, which we described in Chapter 3. Our implementation is an object-oriented approach, based on variation diffs implemented in `DiffDetective` [1, 15]. FeatureSplit is written in Java and the full source code is publicly available on GitHub[1].

We first elaborate on `DiffDetective` in Section 4.1. We discuss the differences between the concept and implementation of a variation diff. Secondly, we describe how we integrate FeatureSplit into `DiffDetective` in Section 4.2. In Section 4.2.1, we discuss our approach to deep-copy variation diffs. Deep copying variation diffs is used in the first step of our operator, to extract atomic diff from the initial variation diff, as described in Section 3.3.1. In Section 4.2.2, we describe our implementation of an equality check between two variation diffs. The comparison of two variation diffs is also used in the first step, to filter duplicates when creating a set of atomic diffs, described in Section 3.3.1. In Section 4.2.3, we describe the implementation of the second step of FeatureSplit and describe possible runtime bottlenecks with our implementation. In Section 4.2.4, we describe how two variation diffs can be combined. The combination of variation diffs is used in the third step of FeatureSplit, as described in Section 3.3.3. Finally, in Section 4.3 we discuss the possible implementation of a feature query generator, which extracts feature queries from a variation diff. The implementation of the feature query generator is based on the concept discussed in Section 3.4 and is required for the second step of FeatureSplit, which is described in Section 3.3.2.

4.1 Overview of DiffDetective

DiffDetective implements a pipe-and-filter architecture, to gather information about a given git repository [15]. DiffDetective reads a given repository, by parsing every patch in a commit from a given repository and creating variation diffs based on each patch. The parsed variation diffs can then be further analyzed. FeatureSplit resides inside the analyzing step of DiffDetective. For this reason, the scope of this thesis will solely cover the design of the variation diff implementation and the analyzing step of DiffDetective.

[1] https://github.com/VariantSync/DiffDetective/tree/thesis_lb_v1

Due to the implementation of DiffDetective in Java, the implementation of variation diffs follows an object-oriented design. In contrast to the concept of a variation diff, the implementation of it uses node objects, where all data about a certain node is stored inside the node itself. Parent and child nodes, for example, are stored references to other node objects inside a node object. This referencing of parent and child nodes is used to build the variation diff. This means, in DiffDetective any node in a variation diff can be used to reference a subtree. The variation diff itself solely acts as a container, which stores some metadata about the whole variation diff and references a subtree variation diff. The variation diff is used as an entry point for variation diff traversal and variation diff altering. The variation diff implementation is called a *DiffTree* and the implementation of a node in a variation diff is described as a *DiffNode*.

The object-oriented implementation of variation diffs allows for an easier generation of the tree structure at the cost of a higher complexity for altering the structure. Because FeatureSplit operates on variation diffs, we had to implement the following solutions to alter variation diffs.

4.2 Implementation of FeatureSplit

Similar to the in Chapter 3 described concept of our operator, the implementation of FeatureSplit is divided into a three-step process. First, the given initial variation diff is divided into atomic diffs, which are variation diff, which cannot be split any further without invalidating the definition of variation diffs. Second, the atomic diffs are grouped into different clusters, depending on the given feature query. Lastly, every atomic diff in a cluster is combined to one feature-aware variation diff representing the cluster.

For the first step of FeatureSplit we discuss the implementation of our variation diff duplication algorithm in Section 4.2.1, and our variation diff comparison algorithm in Section 4.2.2. Because the remaining implementation of the first step of FeatureSplit is similar to the concept, we do not explain it in further detail in the following sections. The second step of FeatureSplit is described in Section 4.2.3. The third step of FeatureSplit is described in Section 4.2.4. Lastly, in Section 4.3 we described our implementation for a feature extraction algorithm.

4.2.1 Deep-Copy Of Variation Diffs

A deep-copy of variation diffs is required for the implementation of the extraction of atomic diffs. To achieve a deep-copy of the object-oriented implementation of a variation diff, we decided to use a hash map as a temporary data structure, instead of traversing the variation diff multiple times to duplicate nodes in the correct order. The temporary hash map is used to quickly access each node in the deep-cloned variation diff, and is created with two complete traversals of the given variation diff. The temporary hash map reduces the time complexity but requires more memory space for storing references to the deep copied nodes.

In Listing 4.1, we show the implementation of the above discussed deep-copy of variation diffs. The hash map, defined in line 2 of Listing 4.1, stores the `id` of each node as the key and the node itself as its value. Our deep-copy algorithm traverses the variation diff two times, the first traversal duplicated the nodes and the second one duplicated

the edges between each node. The first traversal is initiated in lines 30 to 33, where the variables necessary for the duplication are reset and the first traversal is called. The second traversal is initiated in lines 34 and 36, where the boolean indicator of the current traversal is changed before starting the traversal. This is necessary because we can not guarantee that in a single traversal both nodes of a copied edge are copied with it, which possibly would result in a referencing error. The copied edge would have to be temporarily stored until the copied nodes exist. To avoid this complexity we traverse the variation diff twice.

In the first traversal, which is implemented in lines 45 to 49, all nodes from the variation diff are shallow duplicated and added to the temporary hash map. Shallow duplication means the duplication of a node, without its edges (i.e., references to its child or parent nodes).

In the second traversal, which is implemented in lines 52 to 67, the parent-child references of each node are copied and added to the shallow copied nodes in the temporary hash map. For each node in the original variation diff, the `ids` of both parent nodes are retrieved. Then, if the shallow copied node with the same `id` as the inspected original node already has parents, the node is skipped. This is done to prevent redundant linking of nodes, because we add all parent nodes in the first inspection of a node in the second traversal. Otherwise, the shallow copied node is added below the shallow copied parents with the same `id` retrieved earlier. Lastly, in line 20, to extract the deep copied variation diff from the hash map, the root node of the copied variation diff is returned.

```
1  public class Duplication implements DiffTreeVisitor {
2    private HashMap<Integer, DiffNode> duplicatedNodes;
3    private boolean hasAllNodes;
4
5    /**Duplicates a given node
6      *
7      * @return A duplication of the input node without parent and
       ↪ child notes
8      */
9      public static DiffNode shallowClone(DiffNode node) {
10       return new DiffNode(node.diffType, node.codeType, node.
       ↪ getFromLine(), node.getToLine(), node.
       ↪ getDirectFeatureMapping(), node.getLines());
11   }
12
13   /** Tree duplication */
14   public DiffTree deepClone(DiffTree diffTree) {
15       return new DiffTree(deepClone(diffTree.getRoot()), diffTree.
       ↪ getSource());
16   }
17
18   /** Subtree duplication */
19   public DiffNode deepClone(DiffNode subtree) {
20       return deepCloneAsHashmap(subtree).get(subtree.getID());
21   }
22
23   /** Tree duplication which is returned as a hashmap for easier
       ↪ manipulation */
24   public HashMap<Integer, DiffNode> deepCloneAsHashmap(DiffTree
       ↪ tree) {
```

```
25          return deepCloneAsHashmap(tree.getRoot());
26      }
27
28      /** Subtree duplication which is returned as a hashmap for
         ↪ easier manipulation */
29      public HashMap<Integer, DiffNode> deepCloneAsHashmap(DiffNode
         ↪ subtree) {
30          this.duplicatedNodes = new HashMap<>();
31          this.hasAllNodes = false;
32          // fill hashmap
33          DiffTreeTraversal.with(this).visit(subtree);
34          this.hasAllNodes = true;
35          // Add connections
36          DiffTreeTraversal.with(this).visit(subtree);
37          return this.duplicatedNodes;
38      }
39
40      /** Create a shallow clone of every node */
41      @Override
42      public void visit(DiffTreeTraversal traversal, DiffNode subtree)
         ↪  {
43        // Generate nodes
44        if (!this.hasAllNodes) {
45          this.duplicatedNodes.put(subtree.getID(),
46                                   shallowClone(subtree));
47          for (final DiffNode child : subtree.getAllChildren()) {
48              traversal.visit(child);
49          }
50        // create connections
51        } else {
52          for (final DiffNode child : subtree.getAllChildren()) {
53            Integer beforeParentId = child.getBeforeParent() != null ?
         ↪  child.getBeforeParent().getID() : null;
54            Integer afterParentId = child.getAfterParent() != null ?
         ↪ child.getAfterParent().getID() : null;
55
56            if (this.duplicatedNodes.get(child.getID())
57                                   .getBeforeParent() != null ||
58                this.duplicatedNodes.get(child.getID())
59                                   .getAfterParent() != null) {
60              continue;
61            }
62
63            this.duplicatedNodes.get(child.getID()).addBelow(
64              this.duplicatedNodes.get(beforeParentId),
65              this.duplicatedNodes.get(afterParentId));
66            traversal.visit(child);
67          }
68      }
69  }
```

Listing 4.1: Implementation of a variation diff deep-copy

4.2.2 Comparison Of Variation Diffs

In step 1 of FeatureSplit, the extraction of a set of atomic diffs from a variation diff requires a method to compare atomic diffs with each other. To compare two unrelated graphs with each other we would have to solve graph isomorphism for atomic diffs. Because all the compared atomic diffs were extracted from the same initial variation diff it is sufficient for us to rely on the node's ids. Two atomic diffs are equal if and only if the three criteria are met:

1. The root of both atomic diffs has to be identical. In this case it means, the ids of both root nodes have to be identical.

2. Each node in one of the atomic diffs also has to exist in the other atomic diff.

3. The children of identical variation diff nodes in both atomic diff, have to reference identical children nodes.

To compare two atomic diffs for equality, we decided to implement a temporary hash map. As described in Section 4.2.1, a temporary hash map provides a reduction of time complexity in comparison to atomic diff traversal in certain cases. One such case occurs, if we want to access multiple node in a variation diff in a non-sequential order. In such a case we would need to traverse the whole tree for each accessed node if we would only use atomic diff traversal. With a temporary hash-map we need to traverse the variation diff once and have a constant lookup time with the hash-map. This statement is true for the atomic diff comparison because naïvely, for each node in the first atomic diff, the whole other atomic diff has to be inspected to possibly find an identical tree node.

Our implementation, described in Listing 4.2, compares exactly two atomic diffs with each other. Our operator traverses each atomic diff once and adds a reference of the traversed node to the key in the hash map with the same `id` as the traversed node. This process is implemented in lines 13 and 14, which calls the atomic diff traversal described in lines 30 to 38. After populating the hash map, the nodes stored with a similar key are compared with each other. The comparison is implemented in line 17 to 25. To compare two nodes, the nodes themselves and the references to their children are compared. As described above we assume, that atomic diff nodes referenced to the same key in the temporary hash map have to be equal, because the `id` of the two atomic diff nodes have to be similar if they stem from the same initial variation diff. This approach save us the shallow comparison for each node. Finally, after each node pair has been compared, the root nodes of both variation diffs are compared with each other. If every step was successful, both variation diffs are assumed to be equal.

The resulting temporary hash map is visualized in Table 4.1. This table demonstrates, that if a key in the hash map has an odd number of values, both atomic diffs cannot be similar. The table uses for demonstration the first two atomic diff from Figure 3.4 as input. The table demonstrates in row 2 and 3, how the atomic diff comparison identifies that two atomic diffs are not similar. We can also see in row 1 of the table, that we have to inspect the edges of the nodes as well, to ensure, that the nodes and the location of the nodes in the atomic diff are identical.

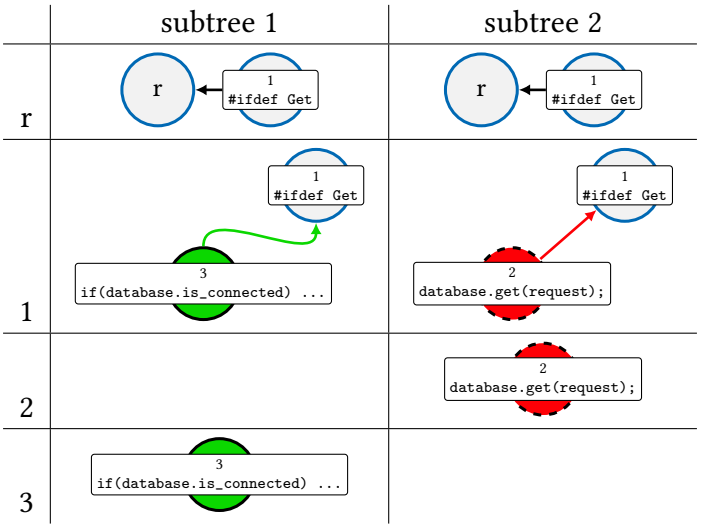


Table 4.1: Temporary hash map of the subtrees 1 and 2, described in Figure 3.4

4.2.3 Variation Diff Clustering

The implementation of the clustering process of variation diffs is described in Listing 4.3. The clustering implementation requires a list of variation diffs and a list of feature queries as input. Contrary to the concept we use a list of feature queries, instead of one feature query. This change lets us extract multiple features at once, without redundantly computing step one and three of FeatureSplit. FeatureSplit is sequentially inspecting every feature query in the feature query list and will cluster an atomic diff to the first matching feature query, removing it from the evaluation of the other feature queries. In line 8, we use a hash map to store the identified variation diffs to their corresponding clusters. The linking of variation diffs to certain clusters is defined from line 11 to line 23. For each variation diff, in line 13 to line 21, each given feature query is evaluated for a match. Which feature queries are evaluated first is determined by the position of the feature query in the feature query list. The evaluation is described in lines 32. In the evaluation, the presence condition of every node's ancestor nodes is inspected and matched with the feature query with a SAT solver. The SAT solver is called in line 41 and based on the in Section 3.3.2.1 describe concept.

Due to the high number of SAT solver calls, some commits with complicated propositional formulas as presence conditions can have extensive computation times.

```java
public class AtomicDiffComparison implements DiffTreeVisitor {
  private HashMap<Integer, List<DiffNode>> comparisonHashMap;

  /** Validate tree equality  */
  public Boolean equals(DiffTree first, DiffTree second) {
    return equals(first.getRoot(), second.getRoot());
  }

  /** validate subtree equality */
  public Boolean equals(DiffNode first, DiffNode second) {
    this.comparisonHashMap = new HashMap<>();
    //populate hash map
    DiffTreeTraversal.with(this).visit(first);
    DiffTreeTraversal.with(this).visit(second);

    // compare elements
    return !this.comparisonHashMap.values().stream()
      .map(diffNodes -> diffNodes.size() != 2
        || !(diffNodes.get(0).getID() == diffNodes.get(1).getID())
        || !diffNodes.get(0).getAllChildren().stream()
            .map(DiffNode::getID).collect(Collectors.toSet())
            .equals(
          diffNodes.get(1).getAllChildren().stream()
          .map(DiffNode::getID).collect(Collectors.toSet())
        )).collect(Collectors.toSet()).contains(true);
  }

  /** Traverses subtrees to populate the comparison hash map  */
  @Override
  public void visit(DiffTreeTraversal traversal, DiffNode subtree)
   ↪  {
    if (!this.comparisonHashMap.containsKey(subtree.getID()))
      this.comparisonHashMap.put(subtree.getID(), new ArrayList
   ↪ <>());
    this.comparisonHashMap.get(subtree.getID()).add(subtree);

    for (final DiffNode child : subtree.getAllChildren()) {
      traversal.visit(child);
    }
  }
}
```

Listing 4.2: Implementation of the variation diff comparison

```java
/**
* Generates a cluster for each given query and
* a "remaining" cluster for all non-selected subtrees
* @param queries: feature mapping which represents a query.
* @return true if query is implied in the presence condition
*/
public static HashMap<String, List<DiffTree>> generateClusters(List<DiffTree> subtrees, List<Node> queries) {
  HashMap<String, List<DiffTree>> clusters = new HashMap<>();
  clusters.put("remains", new ArrayList<>());

  for (DiffTree subtree : subtrees) {
    boolean hasFound = false;
    for (Node query : queries) {
      if (evaluate(subtree, query)) {
        if (!clusters.containsKey(query.toString())) clusters.put(query.toString(), new ArrayList<>());
        // call by reference
        clusters.get(query.toString()).add(subtree);
        hasFound = true;
        break;
      }
    }
    if (!hasFound) clusters.get("remains").add(subtree);
  }
  return clusters;
}

/**
* compares every node in the subtree with the query
* @return true if query is implied in a presence condition of a node
*/
private static Boolean evaluate(DiffTree subtree, Node query) {
  return subtree.anyMatch(node -> (node.getBeforeParent() != null && satSolver(node.getBeforePresenceCondition(), query)) || node.getAfterParent() != null && satSolver(node.getAfterPresenceCondition(), query));
}

/**
* Checks if the query and the presence condition intersect, by checking implication.
* @param query: feature mapping which represents a query.
* @return true if query is implied in the presence condition
*/
private static boolean satSolver(Node presenceCondition, Node query) {
  return SAT.implies(presenceCondition, query);
}
```

Listing 4.3: Implementation of the cluster generator

4.2.4 Variation Diffs Composition

Conceptually, the composition of two variation diffs is the union of all nodes and edges in both variation diffs, which is described in detail in Section 3.3.3. The implementation of this union operation can be achieved by the described operator in Listing 4.4, which realizes a similar behavior with the following approach: First, from line 2 to line 4 the input edge cases are covered, which can be used as a base case for the recursive calling of the composition function. Second, in line 6 all edges in the provided variation diffs are extraced, by calling Listing 4.5. In Listing 4.5, from line 2 to line 6 all nodes from the two variation diffs are extracted and added to a list for simpler access. From the extracted nodes all edges present in the given variation diffs are extracted, defined from line 8 to line 11. The edge class is described in Listing 4.6 and hold two nodes, one parent and one child node. Then, from line 11 to line 21 each edge is inspected. In lines 12 and 13 we add the shallow copy of each node in an edge that has not been already added to the composition variation diff. From line 19 to line 21 we add all edited edges to the nodes in the composed variation diff. Edited edges are edges that were added or removed in one of the given variation diffs. Third, in line 23 the composed nodes are transformed into a variation diff, which will be returned by the composition function. Lastly, before we return the composed variation diff we add the remaining edges of unedited nodes to the composed variation diff. This step is implemented in line 25 to line 37. Additionally, in line 38 we perform a consistency check to ensure, that our returned composed variation diff is valid.

4.3 Feature Query Generator

FeatureSplit requires in addition to a variation diff a queue of feature queries as input. The queue of feature queries can be created manually or generated. Depending on the project FeatureSplit is used in, different feature query generator strategies could be optimal. In this section, we present Listing 4.7, a feature query generator discussed in Section 3.4, which extracts all features from a given variation diff. To acquire the queue of feature queries every node in the passed variation diff is inspected which is described in line 3. For all nodes, that contain a feature mapping, the literals in the feature mapping are extracted and the features in the literals are joined and converted into a set. This process is implemented with lines 4 to 9. In this case, the set of all features is returned which is described in line 11, but in more sophisticated feature query generators the set could be converted into a list and sorted after specific rules.

4.4 Summary

In this chapter, we explained how we implemented FeatureSplit into DiffDetective. We provided an overview of DiffDetective which implements variation trees and variation diffs, and presented the implementation of FeatureSplit and a feature query generator. Based on our implementations, we extended DiffDetective with a variation diff deep-clone function and a non-general variation diff comparison function. In the next chapter, we use our implementation to empirically evaluate the practical significance of our proposed operator.

```
public static DiffTree composeDiffTrees(DiffTree first, DiffTree second) {
  if (second == null && first == null) return null;
  if (first == null || first.isEmpty()) return second;
  if (second == null || second.isEmpty()) return first;

  HashSet<DiffEdge> allEdges = getAllEdges(first, second);
  HashSet<DiffNode> composedTree = new HashSet<>();
  DiffNode composeRoot = DiffNode.createRoot();
  composedTree.add(composeRoot);

  allEdges.forEach(edge -> {
    if (!composedTree.contains(edge.parent)) composedTree.add(Duplication.shallowClone(edge.parent));
    if (!composedTree.contains(edge.child)) composedTree.add(Duplication.shallowClone(edge.child));

    DiffNode cpParent = composedTree.stream().filter(node -> node.equals(edge.parent)).findFirst().orElseThrow();
    DiffNode cpChild = composedTree.stream().filter(node -> node.equals(edge.child)).findFirst().orElseThrow();

    // Add all changes, unchanged node edges aren't added here
    if (cpChild.isAdd() || cpChild.isNon() && cpParent.isAdd()) cpParent.addAfterChild(cpChild);
    if (cpChild.isRem() || cpChild.isNon() && cpParent.isRem()) cpParent.addBeforeChild(cpChild);
  });

  DiffTree composeTree = new DiffTree(composeRoot, first.getSource());

  allEdges.forEach(edge -> {
    if (!edge.child.isNon()) return;

    DiffNode cpParent = composedTree.stream().filter(node -> node.equals(edge.parent)).findFirst().orElseThrow();
    DiffNode cpChild = composedTree.stream().filter(node -> node.equals(edge.child)).findFirst().orElseThrow();

    if (cpChild.getBeforeParent() == null && cpChild.getAfterParent() != null)
        cpChild.getAfterParent().addBeforeChild(cpChild);
    if (cpChild.getAfterParent() == null && cpChild.getBeforeParent() != null)
        cpChild.getBeforeParent().addAfterChild(cpChild);
    if (cpChild.getBeforeParent() == null && cpChild.getAfterParent() == null)
        cpChild.addBelow(cpParent, cpParent);
  });
  composeTree.assertConsistency();
  return composeTree;
}
```

Listing 4.4: Implementation of the variation diff composition

```
private static HashSet<DiffEdge> getAllEdges(DiffTree first,
    ↪ DiffTree second) {
  HashSet<DiffNode> allFirstNodes = new HashSet<>(first.
    ↪ computeAllNodes());
  HashSet<DiffNode> allSecondNodes = new HashSet<>(second.
    ↪ computeAllNodes());
  ArrayList<DiffNode> allNodes = new ArrayList<>();
  allNodes.addAll(allFirstNodes);
  allNodes.addAll(allSecondNodes);

  HashSet<DiffEdge> allEdges = allNodes.stream().map(
    node -> node.getAllChildren().stream().map(
      child -> new DiffEdge(node, child)
    ).collect(Collectors.toSet())).collect(HashSet::new, Set::
    ↪ addAll, Set::addAll);
  return allEdges;
}
```

Listing 4.5: Implementation of variation diff edge extraction

```
public class DiffEdge {
  public final DiffNode parent;
  public final DiffNode child;

  public DiffEdge(DiffNode parent, DiffNode child) {
    this.parent = parent;
    this.child = child;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    DiffEdge diffEdge = (DiffEdge) o;
    return Objects.equals(parent, diffEdge.parent) && Objects.
    ↪ equals(child, diffEdge.child);
  }

  @Override
  public int hashCode() {
    return Objects.hash(parent, child);
  }
}
```

Listing 4.6: Implementation of a variation diff edge

```
public static Set<String> featureQueryGenerator(DiffTree diffTree)
    ↪ {
  HashSet<String> allQueries = new HashSet<>();
  diffTree.forAll(n -> {
      Node formula = n.getDirectFeatureMapping();
      if (formula != null) {
          allQueries.addAll(formula.getLiterals().stream().map(
              literal -> literal.var.toString()
            ).collect(Collectors.toSet()));
      }
  });
  return allQueries;
}
```

Listing 4.7: Implementation of feature query generator

5. Evaluation

In this chapter, we empirically evaluate accuracy and performance of FeatureSplit applied to a dataset of five different open-source repositories listed in Figure 5.2. In our quantitative evaluation, we evaluate to what extent FeatureSplit can extract feature-aware commits from initial commits, and we evaluate what kind of inputs for FeatureSplit produce long computation times and for what reason.

First, we introduce our research questions in Section 5.1. Then, in Section 5.2, we show the in our evaluation used datasets and evaluation environment DiffDetective. In Section 5.3, we present the results of our evaluation. In Section 5.4, we discuss our results and answer our research questions based on our findings. Finally, in Section 5.5, we discuss threats to the validity of our evaluation.

5.1 Research Questions

In our evaluation, we answer two major research questions, subdivided into multiple research questions. The first major question is concerned with the accuracy of FeatureSplit. The second major question is concerned with the scalability of FeatureSplit.

5.1.1 RQ1: Accuracy

The following research questions are concerned with the estimation of accuracy and usefulness of FeatureSplit.

***RQ1.1**: What is the average ratio of feature-aware patches to the initial patches?*

To estimate if FeatureSplit is performing as expected, we track how often FeatureSplit divided an initial variation diff into multiple feature-aware variation diffs. We are measuring this, by tracking the amount of feature-aware variation diffs FeatureSplit generates. If the amount is greater than the amount of the initial variation diffs, our operator is performing as expected. Answering this research question gives us a tendency towards the usefulness of FeatureSplit. The more feature-aware patches are extracted, the higher the reduction of complexity and the increase in usefulness.

***RQ1.2**: What is the typical size of feature-aware patches in relation to the initial patch?*

The follow-up question allows us to determine how intrusive FeatureSplit is when extracting feature-aware variation diffs. We can measure this by averaging the size ratio between initial and feature-aware patches. With this information, we can assess if FeatureSplit creates feature-aware commits of equal size or if FeatureSplit extracts only specific parts of the initial variation diff into a variation diff representing the given feature.

***RQ1.3**: Does FeatureSplit produce any faulty feature-aware patches?*

Lastly, we are interested in potential bugs in FeatureSplit, which could result in FeatureSplit generating invalid variation diffs. To know if the extracted feature-aware variation diffs are valid, we check each extracted feature-aware variation diff for consistency and track the amount of invalid variation diffs.

5.1.2 RQ2: Performance

The following research questions are concerned with the possible limitations of FeatureSplit.

***RQ2.1**: How efficient is FeatureSplit in the extraction of feature-aware patches?*

***RQ2.2**: Does FeatureSplit have any size constraints of a given commit?*

Research question **RQ2.1** and **RQ2.2** allow us to determine to what range of repositories FeatureSplit is applicable to. For our evaluation, we focused on two metrics. First, in **RQ2.1**, we want to measure the average computation time of a feature-aware patch, to determine if FeatureSplit is also applicable to large repositories with many commits. Second, we want to find potential limitations of input values for FeatureSplit. This means, in **RQ2.2**, we inspected possible size limitations of variation diffs passed to FeatureSplit. We define the size of a variation diff as the amount of nodes a given variation diff has.

5.2 Study Design

On the basis of five open-source repositories we further displayed in Figure 5.3, we empirically evaluate FeatureSplit. All repositories used for evaluation are publicly available on GitHub and use the C-preprocessor as its software product line implementation. For our evaluation, we use DiffDetective[1] [15] as our evaluation environment. We modified the existing evaluation environment and created a custom environment with a clone-and-own approach. DiffDetective converts the commits in repositories into variation diffs, which are passed to FeatureSplit for our evaluation. To interact with the feature

[1] https://github.com/VariantSync/DiffDetective

queries, FeatureSplit uses Sat4J[2] [23] to evaluate the propositional formula of a feature query. Additionally, FeatureSplit uses Functjonal[3] to implement certain functions with functional programming. It is important to note, that we do not inspect merged commits and commits with ill-formed C-preprocessor annotations. This results in fewer processed commits than available in a repository.

Our evaluation process is visualized in Figure 5.1. We first clone every repository. Second, for each repository, we extract all variation diffs using DiffDetective. Third, for each variation diff we extract all features used in the variation diff. For this step, we utilize the feature query generator described in Section 3.4, which retrieves all features in a given variation diff. Fourth, FeatureSplit converts each variation diff to feature-aware variation diffs based on the features extracted with the feature query generator. Last, each extracted evaluation data point is stored and quantitatively evaluated afterwards. We discuss the values stored in the evaluation data point in the next paragraph.

For research question **RQ1.1**, we track the ratio of extracted feature-aware and remainder variation diffs to the initial variation diffs of each repository. We achieve this by averaging the amount of feature-aware and remainder variation diffs generated from each initial variation diff. To answer **RQ1.2**, and part of **RQ2.2** we track the size of each initial variation diff, the size of the resulting feature-aware variation diff, and the size of the resulting remainder variation diff. The size of a variation diff represents the number of nodes a given variation diff has. To answer if FeatureSplit produces any faulty feature-aware variation diffs (**RQ1.3**), we evaluate the validity of each generated feature-aware variation diff and track the number of times the validation failed. For research questions **RQ2.1** and **RQ2.2**, we measure the performance of our operator with the following evaluation results. We track the process times of a repository, the process times of individual commits, the number of commits in a repository, and the already described sizes of variation diffs in a repository. To answer research question **RQ2.1**, we compare the number of processed commits with the required processing time. For **RQ2.2**, we track the largest variation diff and compare it with the processing time of its commit. This comparison allows us to evaluate if the variation diff size is a restricting factor for our operator.

To prevent not terminating evaluations of repositories, we track intermediate evaluation results of FeatureSplit and our evaluation includes a runtime threshold to stop the evaluation if it exceeds the threshold. The runtime threshold is based on the evaluation results presented by Bittner et al. [12]. To avoid penalization of larger repositories with more commits, we calculate the evaluation time of each repo relative to its total number of commits with Equation 5.1:

$$\text{runtime threshold} = 10000ms + (\text{number of commits} \times 50ms) \tag{5.1}$$

We perform our validation on a Windows 10 system with 64-bit architecture and an AMD Ryzen 5 3600X CPU with 3800Mhz clock rate. Our system has 12 threads that were used in parallel to evaluate FeatureSplit, where a batch of 1,000 commits allocates a single thread. The system was not used for other purposes during our evaluation.

[2] https://gitlab.ow2.org/sat4j/sat4j
[3] https://github.com/VariantSync/Functjonal

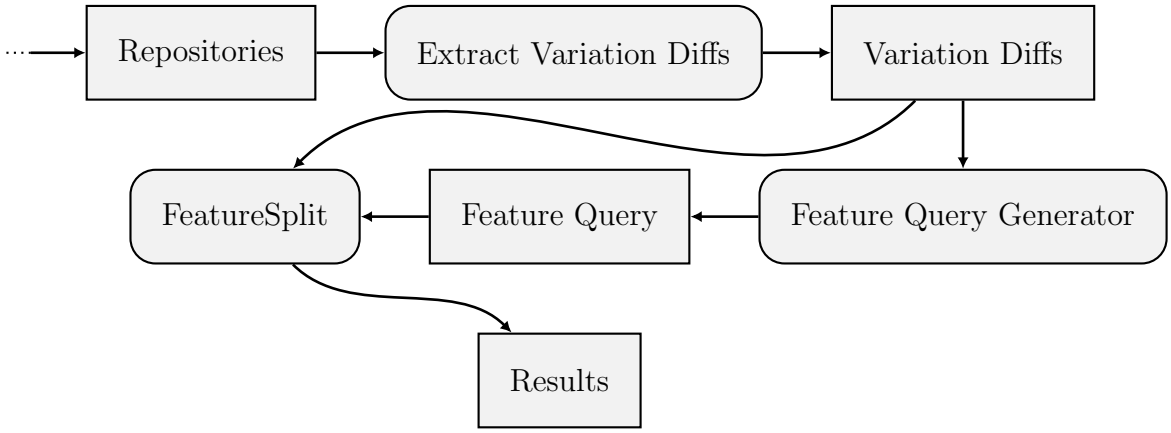


Figure 5.1: Evaluation procedure

5.2.1 Dataset

Our evaluation results are extracted from five open-source git repositories. We chose these five repositories from a set of repositories used by Bittner et al. [12] for the validation of variation diffs and their classification. The criteria for our selection process are to include repositories from a broad range of total number of commits, repository sizes, and domains. The selected repositories are listed in Figure 5.2 and cover different domains, like operating systems, databases, network clients, e-mail clients, or servers. The amount of total commits range from 7 to 1,072,142 and the number of lines of code range from 29 thousand to around 28 million.

Project name	Domain	Total commits	Lines of code
berkeley-db-libdb	database system	7	508,000
sylpheed	e-mail client	2,682	525,000
libssh	network	5,349	139,900
apache-httpd	web server	32,927	29,000
linux	operating system	1,072,142	27,800,000

Figure 5.2: General information about the evaluated repositories. Further details in Figure A.1

Project name	Completed	Processed commits	Patches	Features	Calculation Time
berkeley-db-libdb	yes	1	485	72	2.59s
sylpheed	yes	1,820	4,239	174	46.62s
libssh	yes	4,562	8,791	358	97.13s
apache-httpd	runtime threshold reached	15,253	32,881	1,696	27min
linux	parser error	16,568	31,910	3,977	6.3min

Figure 5.3: General evaluation results

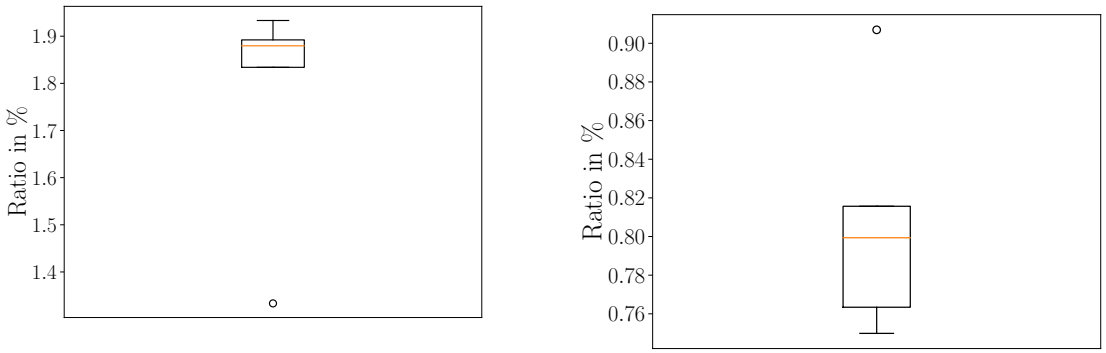
5.3 Results

To provide a general overview over our evaluation results, we provide Figure 5.3, which displays general information about the repositories we analyzed. The *Completed* column displays whether the evaluation was completed successfully, or if it was terminated due to errors, bugs, or exceeding the runtime threshold. In the *Processed commits* column, we count the number of commits that were evaluated by our operator. The *Patches* column describes the number of patches which were converted to variation diffs and used in FeatureSplit. The *Features* column shows the number of features contained within a repository. Finally, the *Calculation time* shows the evaluation time for each given repository. If a repository reached the runtime threshold, which is displayed in the *Completed* column, the evaluation time is equal to the time restrictions described, in Equation 5.1.

Our first result is plotted in Figure 5.4(a), where the ratio of feature-aware patches to the initial patches of each repository is displayed. We use this figure to answer **RQ1.1**. To visualize our evaluation results, we use a boxplot, which allows us to display the evaluation results of all repositories to one research question within one diagram. The yellow line in the bloxplot showcases the average in the displayed data. The top line of the box displays the upper quartile and the bottom line of the box displays the lower quartile. Horizontal lines above or below the box display the minimum and maximum values of our data, and circles represent outliers. Even though the rate of feature-aware patches to initial patches fluctuates depending on the repository, the average amount of feature-aware patches is 88% higher than the average amount of initial patches. Our results show a top quartile of 90% and a bottom quartile of 82% of more feature-aware patches. The repository representing the bottom outlier still extracts feature-aware patches 35% of the time.

Second, in our evaluation, we track the ratio of size decrease between initial variation diffs and feature-aware variation diffs. In Figure 5.4(b), we display the ratio of initial variation diff sizes to feature-aware variation diff sizes over all repositories. We use this figure to answer **RQ1.2**. The average size reduction lies between 18 and 23 percent, with a maximum of 24% and an outlier of 4%.

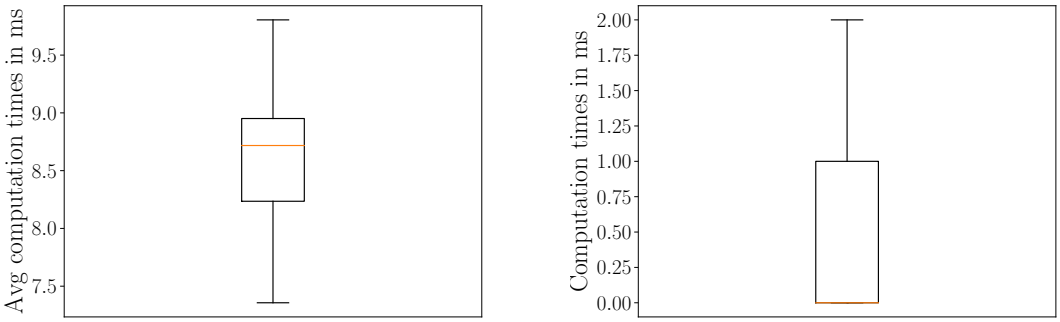
Third, we trace the number of ill-formed feature-aware patches produced by our operator. Over all repositories we inspected, none extracted feature-aware variation diffs were ill-formed. These results are used to answer **RQ.1.3**.



(a) Ratio of feature-aware patches (b) Ratio of feature-aware variation diff sizes

Figure 5.4: Evaluation Results displaying the accuracy of FeatureSplit

Fourth, in Figure 5.5(a) we track the average computation times of patches with our operator in milliseconds. The average computation time per patch required 8.75ms, with a bottom quartile of 8.25ms and a top quartile of 9ms. These results are related to **RQ2.1**.



(a) Average computation times of FeatureSplit in ms (b) Process times of the largest patches in ms

Figure 5.5: Evaluation Results displaying the performance of FeatureSplit

Lastly, we show the computation times of the largest patches Figure 5.5(b), which are related to **RQ2.2**. The computation times of the largest patches range from 0ms to 1ms, with one outlier at 2ms. We cannot obtain more precise results, because Java cannot perform smaller time measurements.

5.4 Discussion

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

5.4.1 RQ1.1: Average Ratio of Feature-Aware Patches

The average ratio between feature-aware variation diffs and initial variation diffs, is an estimation of FeatureSplit's complexity reduction capabilities. With an average result of

88% more feature-aware patches than initial patches, we argue that FeatureSplit can indeed reduce the complexity of software product line repositories. By the mere fact, that 88% of initial patches were reduced in size due to FeatureSplit, we argue that the size reduction is a strong indicator for the usefulness of FeatureSplit. To evaluate whether the complexity reduction also reduces the complexity for developers and increases productivity, our operator has to be further tested in a user study by integrating FeatureSplit into developing processes and measure the usage of FeatureSplit, by multiple developers. An additional insight we gained with this result, is that most commits in the tested repositories are not feature-aware, which means composite commits seem to be a regular occurrence and FeatureSplit was able to identify and simplify these composite commits.

> **RQ1.1**: On average, FeatureSplit extracted feature-aware patches from 88% of commits.

5.4.2 RQ1.2: Typical Size of Feature-Aware Patches to Initial Patches

Our evaluation results show, that the average size of feature-aware variation diff sizes are around 18 to 23 percent smaller than the initial variation diff sizes. This evaluation result also indicates, that FeatureSplit can be useful for reducing complexity in software product lines.

> **RQ1.2**: The typical size of feature-aware patches in relation to the initial patches is 18% to 23% percent smaller.

5.4.3 RQ1.3: Number of Faulty Feature-Aware Patches

As described in Section 5.3, our operator did not return ill-formed feature-aware variation diffs throughout the whole evaluation. For this reason, we assume that our operator is valid and does not extract ill-formed variation diffs. Further evidence is given by valid unit-tests we implemented to cover edge cases that could occur in FeatureSplit.

> **RQ1.3**: FeatureSplit does not produce any faulty commits in the analyzed dataset histories.

5.4.4 RQ2.1: Efficiency of FeatureSplit

Our evaluation results indicate that the number of patches in a repository are not the main contributor to the required evaluation time. This statement is based on Figure 5.5(a), which displays that, independent of the repository size, the average patch time centered around 9ms. Upon manual inspection we have seen that, even though the size of the repository did not seem to directly affect the computation time of FeatureSplit, complex propositional formulas in a patch increase the computation time greatly. We argue, the reason only large repositories had extensively long computation times for certain complex propositional formulas is based on the fact that larger repositories contain more

patches, increasing the likelihood of finding a patch with a too complex propositional formula. Based on our manual inspection, the solving of complex propositional formulas is the largest computation time contributor in our evaluation, which will be discussed in the next section in more detail.

> **RQ2.1**: The efficiency of FeatureSplit depends on the size of propositional formulas in feature mappings.

5.4.5 RQ2.2: Limitations to Variation Diff Sizes

We noticed, that the size of the initial variation diffs is only affecting the performance of FeatureSplit marginally. Our statement is based on Figure 5.5(b) which shows no relation to the size of a variation diff and its computation time. In many instances, the computation times of large patches were lower than the average computation time of patches in its repository. The major contributor to the computation time of FeatureSplit is the solving of complex propositional formulas. The solving of a propositional formula is required to identify if a variation diff contains a given feature query. As described in Section 4.2.3, the unoptimized and frequent call to solve propositional formulas leads to evaluation times which exceeded the given runtime threshold. Additionally, DiffDetective did not use the available computational power optimally which lead to phases in the evaluation where the evaluation only used a small amount of the available computational power. This lead to increased computation times for some repositories regardless of the computation complexity. For this reason, we argue that with a change or optimization of the used SAT-solver and an optimization of FeatureSplit and DiffDetective, FeatureSplit could also be applied to repositories with complex propositional formulas, without high computation times.

> **RQ2.2**: FeatureSplit does not have input size constraints within the tested datasets.

5.5 Threats to Validity

In the previous section, we measured the accuracy and performance of FeatureSplit. However, there are some threats to the validity of our evaluation.

First, due to the modest amount of evaluated repositories, which all are written in C and C++ and use the C predecessor, we can only assume the results apply to other repositories as well. Additionally, the evaluated repositories were not chosen randomly, this potentially could introduce bias into the evaluation. For this reason we cannot be certain, that FeatureSplit is applicable to other repositories, which use other languages, for example. We minimized this threat, by selecting a set of repositories with a large range in the number of commits. To improve validity, the evaluation could be broadened to more repositories.

Second, the commits which required the most time to compute could not be represented in the evaluation results, due to the implemented runtime threshold. This could mean, that FeatureSplit has a higher calculation time than estimated in our evaluation. The

runtime threshold was necessary to avoid excessively long calculation times for evaluations with complex propositional formulas.

Third, FeatureSplit and the tool DiffDetective which we used to generate variation diffs for our evaluation of FeatureSplit could have bugs that impact our results. In the evaluation of FeatureSplit we detected some errors in DiffDetective which leads us to believe some uncertainties are still existent in the evaluation of variation diffs. It follows, that uncertainties in the evaluation of variation diffs affect the evaluation of FeatureSplit. We minimized the impact of this threat by only analyzing the evaluation data generated before bugs occurred.

5.6 Summary

In this chapter, we evaluated FeatureSplit empirically with respect to five different open-source repositories. We conclude that FeatureSplit can reduce the complexity in commits and that FeatureSplit is applicable to software product line repositories that use the C predecessor and are programmed in C or C++. We also conclude that the computation time of FeatureSplit can grow significantly if complex propositional formula are present in a repository. This especially affects larger repositories, because larger repositories have a higher chance to contain such complex propositional formulas. Even though some repositories contain commits which lead to exceedingly long computation times, we argue that these computation times could be reduced significantly if FeatureSplit would reduce the necessary SAT-calls to cluster atomic diffs and if FeatureSplit would switch or improve the used SAT-solver. Lastly, our results suggest that FeatureSplit does not have any commit size restriction, which could limit the use of FeatureSplit in a real-world environment.

6. Related Work

In this chapter, we discuss related work. First, we compare the existing graph representations of commit differences in Section 6.1. Then, we discuss the current state of variability support in version control systems in Section 6.2. Lastly, in Section 6.3, we discuss variation control systems and their correlation to FeatureSplit.

6.1 Graph Representation of Commit Differences

Various papers suggest, that the graph representation of commit differences are a viable way of splitting composite commits into smaller more manageable
chunks [15], [11], [13], [20], [14]. Various papers independently developed some variation of a graph representation of a commit or a commit difference [15], [11], [20], [14]. SmartCommit uses a graph representation to transform commits into activity-oriented commits [11]. Activity-oriented commits represent one of four types: feature addition, refactoring, adding test, and reformatting. This procedure allows the developer to better understand the actions taken in each commit, but neglects features in software product lines. Janke and Mäder describe a graph-based mining method to gather code change patterns from a version control system repository. Code change patterns illustrate if changes represent an insertion, deletion, modification or movement of source code. The inspection of code change patterns allows different insights than the activity-oriented commits, by focussing on the evolution of the repository rather than the activity of the developer. The graph-based mining method also operates without regard to features implemented in software product lines Finally, Zhou et al. describe their tooling *INFOX* [14], which creates a tree representation of source code to identify non-merged features in forks. This graph representation identifies features of software product lines and generates an overview of the project to inform developers what has happened in each fork. Because *INFOX* does not provide source code separation tooling, the extracted features from *INFOX* could be used in combination with FeatureSplit to automatically create feature-aware commits. Due to the inability of the above-mentioned operators to represent or model features in commits from software product lines, we chose variation diffs introduced in [1] to be the graph representation of commits for our operator FeatureSplit. In comparison to other graph representations, variation diffs are complete

and sound. A variation diff is graph representation of a commit, which integrates the according feature mappings of source code into its graph representation.

6.2 Variability in Version Control Systems

Due to the only partially supported software variability in commonly used version control systems, papers have been published to provide better software variability support in commonly used version control systems. Herzig and Zeller provided further insight [5], that current designs of version control systems tempt developers to combine bug fixes from different features into entangled (composite) commits, out of convenience. This insight was a driving factor to develop FeatureSplit. The tool *INFOX* [14], already discussed in Section 6.1, resolves this issue for uncommitted commits, but does not improve already committed entangled commits. One attempt to eliminate composite commits is proposed by Muylaert and De Roover. They discuss the elimination of composite commits by slicing committed changes from the composite commit, depending on the relative proximity of these changes in the abstract syntax tree. The advantage with this approach is the automatic generation of untangled commits. The drawback with this approach in comparison to FeatureSplit is, that a developer does not have fine-grained control over the creation of these untangled commits. Additionally, the generated untangled commits are not feature-aware and the creation of feature-aware commits with complex feature queries is not achievable in the program slicing method either. The inability to cluster the untangled commits into feature-aware commits [4], was acknowledged and solved by Dias et al. with their tool *EpiceaUntangler*. The automatic clustering is performed with a machine learning algorithm. The difference to FeatureSplit is that the *EpiceaUntangler* requires an IDE plugin to track the activity of a developer, which is used for the untangling process. FeatureSplit does not require information about the developers' activity for its feature-aware commit generation. Barnett et al. provided the tool *ClusterChanges*, which also automatically decomposes commits into independent partitions. These partitions can be compared to feature-aware commits produced by FeatureSplit. The difference between *ClusterChanges* and FeatureSplit is that *ClusterChanges* requires the ability to parse and semantically understand the source code which FeatureSplit does not require. Such information required for *ClusterChanges* has to be provided by the used programming languages. Additionally, *ClusterChanges* needs further optimization to avoid false negatives when partitioning commits. Finally, Dintzner et al. [13] developed *FEVER* which analyses the history of software product lines automatically. The analysis extracts data and metadata for co-evolution of variability. This analytical data could be used to extend the capabilities of FeatureSplit, for example better feature extraction methods could be developed based on the gathered data or the clustering process of atomic diff could be optimized. The analytical data gathered though *FEVER* could result in more efficient and accurate feature-aware commits.

6.3 Variation Control Systems

Version control systems are inherently designed for single variant development. Current research including our thesis tries to evolve version control system to allow for better variability support [1] [12] [14] [13] [16]. Issues, that variation control systems try to eliminate are for example that preprocessors in software product lines greatly reduce the

readability of the source code, the larger the software product line becomes [24]. This leads to greater difficulty implementing new code and maintaining the current code base. For this reason variation control systems were developed, which increase readability in the development process, by hiding all source code of features, the developer is not working on [25] [24]. Variation control systems also have other functionality to improve the development of software product lines, like automatic presence condition generation and updating [24]. Even tough variation control systems seem promising, their widespread adoption is still lacking [24]. Reasons for the slow or stagnant adoption of variation control systems, are for example high *cognitive complexity* needed to craft feature mappings, which provide the advantage of only interacting with source code relevant the developed feature. Also, a *high adoption barrier* and after the adoption, a *locked-in syndrome* are major problems, due to difficulty of switching back to a version control system and the proprietary software used in some variation control systems [24]. Even though *cognitive complexity* is a side effect FeatureSplit might share as well, when the developer creates their own feature queries. Other problems like an adoption barrier and cognitive complexity can be avoided, because FeatureSplit can act as an operator between a developer who creates composite commits and a variation control system. This means FeatureSplit would weaken the requirements a variation control system puts on developers. With such an approach developers do not need to change their commit behavior in any way while benefitting from the usage of a variation control system.

7. Conclusion

The current design of version control systems does not prevent developers from combining unrelated changes into composite commits, out of convenience. This behavior can raise the complexity of software product line repositories due to developers needing to mentally separate unrelated from related changes to understand changes to specific features in a commit. To reduce the complexity of the commit history of software product line repositories, we conceptualized and empirically evaluated FeatureSplit, an operator that extracts changes from a given commit that are related to a certain feature query or feature interaction. Therefore, a feature query, is a propositional formula representing a desired combination of features. We implemented the concepts of our operator in DiffDetective, which extracts variation diffs, a graph-based representation of patches from commits, which are called variation diffs.

FeatureSplit follows a three-step process to extract feature-aware variation diff from the initially provided variation diffs. First, the provided variation diff is split into atomic diffs which cannot be separated further without invalidating the graph representation of the patch. Second, the atomic diffs are grouped by a given feature query, which defines which features should be extracted. Lastly, each grouped cluster of atomic diffs is composed into one feature-aware variation diff. Our evaluation shows, that FeatureSplit is able to consistently reduce the complexity of the commit history of software product line repositories.

We envisioned three applications of FeatureSplit in software product lines. For the first application of FeatureSplit we propose, automatically reduces complexity of local commits before a push-request to a remote repository. In concept, local commits are restructured with FeatureSplit into feature-aware commits automatically or by providing one or multiple feature querys for the restructurisation. This would allow developers to create feature-aware and simpler commits without altering the commit habits of a developer. The second application of FeatureSplit would allow for a feature-aware interaction with software product line repositories. Already existing commits could be converted into feature-aware commits with FeatureSplit retroactively. These feature-aware commits could be used to visualize changes made to a single feature or feature formula and to extend the capabilities of current version control systems. Lastly, FeatureSplit could be

a key technology to lower the requirements for variation control system for developers, by allowing develops to keep creating composite commits and still gain the benefits of a variation control system which requires feature-aware commits.

8. Future Work

Our thesis establishes a basis for feature-aware commit extraction and further research in feature extraction operators. Our developed feature extraction operators FeatureSplit is an initial prototype with technical limitations and many opportunities for future work. The main restricting factors of FeatureSplit can be improved with a more efficient usage of SAT-solvers and an improvement in the parser used to convert patches into a graph-based structure. In the following section, we want to address limitations of FeatureSplit and succeeding research questions which should be further illuminated.

8.1 Feature Query Generator

For the evaluation of FeatureSplit, we implemented the feature query generator described in Section 3.4. This feature query generator retrieves all features in a given patch. For our use-cases, our implemented feature query generator is sufficient. For example, it can be used to allow developers to select a feature from all present features in a commit, which the developer can use to request a feature extraction with FeatureSplit and the selected feature. In other cases, a more sophisticated implementation of a feature query generator would be advantageous. This is true, for example, in cases where a developer wants to extract less specific features which are easier to understand for selection of a feature for the feature extraction. One example of a more sophisticated feature query generator would be to combine the feature query generator described in Section 3.4 with the feature model of a software product line, described in Chapter 2. If two features extracted from the feature query generator are closely related to each other in the feature model, the ancestor of the two features could be used to resemble both features in a feature-aware commit. Such an approach would allow for flexibility in the size of the extracted feature-aware commits. Additionally, more sophisticated feature query generators could detect feature queries that produce empty feature-aware commit or detect illegal propositional formulas.

8.2 Customization of Feature Query Evaluation

In FeatureSplit, in the atomic diff clustering step, we use an evaluation function described in Section 3.3.2.1, to evaluate if a variation diff should be included in a feature-

aware cluster or in the remainder cluster. The evaluation function described resembles an optimistic feature query evaluation, because the presence condition of the evaluated variation diff node is stronger than the feature query. Depending on the use-case other feature query evaluation strategies might be advantageous.

In case we want to extract only atomic diffs which have a presence condition that exactly matches the feature query, we can integrate the following evaluation function for the feature query evaluation:

$$\text{evaluateQuery}(v, \Theta) := PC(v) = \Theta \tag{8.1}$$

Lastly, if a more conservative approach for the evaluation of variation diffs is needed, the following feature query evaluation could be used. This feature query evaluation is only satisfied if the stronger feature query is satisfied as well.

$$\text{evaluateQuery}(v, \Theta) := \Theta \models PC(v) \tag{8.2}$$

The above discussed feature query evaluation strategies are not the full set of possible feature query evaluation strategies but can be used in combination to cover all possible evaluation strategy outputs. In future works the above discussed feature query evaluation strategies could be implemented, by allowing developers to pass a specific feature query evaluation to FeatureSplit. To implement the show feature query evaluation strategies, the desired function just has to be passed down to the evaluation function in FeatureSplit, which then replaces the fallback feature query evaluation function.

A. Appendix

Project name	Domain	Repository URL	Clone URL	Estimated number of commits
berkeley-db-libdb	database system	https://github.com/berkeleydb/libdb	https://github.com/DiffDetective/libdb.git	7
sylpheed	e-mail client	https://github.com/jan0sch/sylpheed	https://github.com/DiffDetective/sylpheed.git	2,682
libssh	network	https://gitlab.com/libssh/libssh-mirror	https://github.com/DiffDetective/libssh.git	5,349
apache-httpd	web server	https://github.com/apache/httpd	https://github.com/DiffDetective/httpd.git	32,927
linux	operating system	https://github.com/torvalds/linux	https://github.com/DiffDetective/linux.git	1,072,142

Figure A.1: Evaluated dataset based of dataset used by Bittner, Tinnes, Schultheiß, Viegener, Kehrer, and Thüm [1]

Bibliography

[1] Paul Maximilian Bittner, Christof Tinnes, Alexander Schultheiß, Sören Viegener, Timo Kehrer, and Thomas Thüm. Classifying Edits to Variability in Source Code. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering*, November 2022. To appear. (cited on Page v, vii, 2, 4, 5, 9, 10, 11, 12, 13, 17, 22, 23, 45, 46, and 54)

[2] Thomas Thüm. Software product lines, April 2022. URL https://www.uni-ulm.de/en/in/sp/teaching/software-product-lines/. Lecture. (cited on Page vii, ix, 4, and 5)

[3] Roberto Erick Lopez-Herrejon and Alexander Egyed. Detecting inconsistencies in multi-view models with variability. In *European Conference on Modelling Foundations and Applications*, pages 217–232. Springer, 2010. (cited on Page vii, ix, 4, and 5)

[4] Ward Muylaert and Coen De Roover. [research paper] untangling composite commits using program slicing. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 193–202, 2018. doi: 10.1109/SCAM.2018.00030. (cited on Page 1, 2, and 46)

[5] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 121–130, 2013. doi: 10.1109/MSR.2013.6624018. (cited on Page 1 and 46)

[6] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 341–350, 2015. doi: 10.1109/SANER.2015.7081844. (cited on Page 1, 2, and 46)

[7] Krzysztof Czarnecki. Variability in software: State of the art and future directions. In *International Conference on Fundamental Approaches to Software Engineering*, pages 1–5. Springer, 2013. (cited on Page 1)

[8] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software product line engineering: foundations, principles, and techniques*, volume 1. Springer, 2005. (cited on Page 1 and 3)

[9] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. Feature-oriented software evolution. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, pages 1–8, 2013. (cited on Page 1)

[10] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 134–144, 2015. doi: 10.1109/ICSE.2015.35. (cited on Page 2 and 46)

[11] Bo Shen, Wei Zhang, Christian Kästner, Haiyan Zhao, Zhao Wei, Guangtai Liang, and Zhi Jin. Smartcommit: a graph-based interactive assistant for activity-oriented commits. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 379–390, 2021. (cited on Page 2, 9, and 45)

[12] Paul Maximilian Bittner, Alexander Schultheiß, Thomas Thüm, Timo Kehrer, Jeffrey M. Young, and Lukas Linsbauer. Feature trace recording. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1007–1020, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3468531. URL https://doi.org/10.1145/3468264.3468531. (cited on Page 2, 37, 38, and 46)

[13] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. Fever: An approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. *Empirical Software Engineering*, 23(2):905–952, 2018. (cited on Page 2, 45, and 46)

[14] Shurui Zhou, Stefan Stanciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wasowski, and Christian Kästner. Identifying features in forks. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 105–116. IEEE, 2018. (cited on Page 2, 9, 45, and 46)

[15] Sören Viegener. Empirical Evaluation of Feature Trace Recording on the Edit History of Marlin. Bachelor's thesis, Germany, April 2021. URL https://oparu.uni-ulm.de/xmlui/bitstream/handle/123456789/38679/BA_Viegener.pdf. (cited on Page 2, 9, 23, 36, and 45)

[16] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-oriented software product lines*. Springer, 2016. (cited on Page 4, 5, and 46)

[17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering software variability with FeatureIDE*. Springer, 2017. (cited on Page 4)

[18] Scott Chacon. Git internals. *Pro Git*, pages 223–250, 2009. (cited on Page 6)

[19] Linus Torvalds, Junio Hamano, C, Scott Chacon, and Jason Long. gitdiff documentation. URL https://git-scm.com/docs/git-diff. (cited on Page 6)

[20] Mario Janke and Patrick Mäder. Graph based mining of code change patterns from version control commits. *IEEE Transactions on Software Engineering*, 48(3):848–863, 2022. doi: 10.1109/TSE.2020.3004892. (cited on Page 9 and 45)

[21] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. Feature-to-code mapping in two large product lines. In *SPLC*, pages 498–499. Citeseer, 2010. (cited on Page 11)

[22] Uwe Schöning. *Logik für Informatiker*. Springer, 1987. (cited on Page 12)

[23] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):59–64, 2010. (cited on Page 37)

[24] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. A classification of variation control systems. *ACM SIGPLAN Notices*, 52(12):49–62, 2017. (cited on Page 47)

[25] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. Concepts, operations, and feasibility of a projection-based variation control system. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 323–333. IEEE, 2016. (cited on Page 47)

Declaration of Authorship

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

Place, Date of Submission

Signature