Explaining Edits to Variability Annotations in Evolving Software Product Lines

Lukas Güthing
lukas.guething@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Paul Maximilian Bittner
paul.bittner@uni-ulm.de
Ulm University
Ulm, Germany

Ina Schaefer
ina.schaefer@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Thomas Thüm
thomas.thuem@uni-ulm.de
Ulm University
Ulm, Germany

ABSTRACT

Software is subject to changes and revisions during its development life cycle. For configurable software systems, changes may be made to functionality of source code as well as variability information such as code-to-feature mappings. To explain how code-to-feature mappings change in edits made to configurable software, we relate the mappings before and after an edit in terms of the sets of variants they denote. We prove our explanations to be complete and unambiguous, meaning that every pair of code-to-feature mappings is explained in terms of exactly one relation. Based on a graph formalism, we provide an algorithm for fast detection of relations during commits to version control. In an initial study, we detect relations between feature annotations in 42 real-world software product-line repositories to better understand typical changes in the evolution of configurable software. We demonstrate that our formalism can be automated and that analyzing a commit requires only 135 ms on average.

CCS CONCEPTS

• **Software and its engineering** → **Software evolution**; **Software product lines**; **Preprocessors**.

KEYWORDS

software evolution, software product lines, software variability

ACM Reference Format:
Lukas Güthing, Paul Maximilian Bittner, Ina Schaefer, and Thomas Thüm. 2024. Explaining Edits to Variability Annotations in Evolving Software Product Lines. In *18th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2024), February 07–09, 2024, Bern, Switzerland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3634713.3634725

VaMoS 2024, February 07–09, 2024, Bern, Switzerland

ACM ISBN 979-8-4007-0877-0/24/02
https://doi.org/10.1145/3634713.3634725

1 INTRODUCTION

The defining property of a configurable software system is that the system can be shipped in different forms, so-called *variants*, to for example target different platforms or customize for different custormer requirements. In software product-line engineering, variants are distinguished in terms of *features*. A feature represents an atomic *commonality* [55] between variants implementing that feature, as well as dissimilarity for variants that do not share the feature. To compose features, developers have to document and maintain *code-to-feature mappings*, for example by annotating the code base [5, 29].

While source code itself can be edited, changes to software product lines can also affect variability information, including feature mappings [11, 19, 24, 34]. Next to bugs in functionality, changes to variability information may produce variability bugs [1], for example by invalidating a variant's syntax [29], or altering the set of available variants in an unintended way [30, 51, 60, 64]. Keeping track of changes to code as well as to feature mappings therefore is crucial in configurable software environments.

To understand the evolution of configurable software, an insight into the evolution of feature mappings with respect to their interdependencies is necessary. However, to the best of our knowledge, the evolution of feature mappings themselves is an open field. Evolution of source code has been examined extensively because it is relevant for both configurable as well as single-variant software but is unaware of variability [16, 50, 65, 66, 69]. Kröher et al. [35] inspect changes to functionality and variability over the development history of the Linux kernel but examine *that* but not *how* the variability information before and after an edit is related. Classifications of edits to variability information assess the impact of changes to variability on the source code but changes to feature mappings are modelled naively as insertions and deletions [10, 63]. Relations between deleted and added feature mappings remain opaque. Similarly, Dintzner et al. [19] classify edits to feature mappings in source code as either *added*, *removed*, or *modified* which does not explain how the sets of variants described by the changed feature mappings are related. Further work on the evolution of problem-space variability information [23, 62, 67] explains edits to feature models but not to variability annotations.

In this work, we provide a graph-based formalism to explain the evolution of feature mappings by relating edited feature mappings. Our goal is to provide formal foundations to explain edits to

feature mappings technically such that explanations can be leveraged by analyses and tools. We propose *Implication*, *Alternative*, and *Independent* as binary relations between feature mappings to gain insight into how sets of variants described by changed feature mappings alter. These variant set relations are complete and unambiguous, meaning that we can classify every pair of feature mappings into exactly one of the proposed relations. In summary, our contributions are:

- **Relating Feature Mappings.** We formalize a set of meaningful relations between feature mappings and prove the set's completeness.
- **Formalism.** We propose a graph-based formalism based on variation diffs [10] to explain edits to feature mappings during software product-line evolution.
- **Feasibility Study.** We evaluate our explanations on real-world software product lines in terms of computational effort and the amount of information gained.

2 BACKGROUND

There are several different techniques for implementing software product lines in terms of reusable features which are composed to variants [5, 13, 18, 54], typically distinguished into *annotative* and *compositional* methods [29]. Compositional methods implement features in distict modules which are composed to form variants, such as in plug-in frameworks [28], feature-oriented [6], or aspect-oriented programming [4, 32]. In annotative approaches, features are implemented in a common code base, and annotated with feature information, known as *feature mappings*, to conditionally include or exclude code chunks [3, 5, 20, 21, 29, 33, 49, 53]. Such a feature annotation is typically specified in terms of formula in a propositional or first-order logic. While in the following, we focus on annotative product lines, our reasoning also applies compositional methods to some extent as we will discuss later.

2.1 Annotation-Based Software Product Lines

A common annotative technique to implement annotative software product lines are preprocessors [5, 21, 29] such as the C preprocessor used in many real-world systems [40]. With the C preprocessor, code is mapped to features on a fine-grained level of lines of code or even single characters. An annotation starts with an `#if` annotation followed by a formula containing features as variables, and ends with an `#endif` annotation. As an example, the function `f` in Listing 2.1 is annotated with the feature *A*. Annotations might be nested, in which case all conditions have to be fulfilled for the innermost code fragments to be present. Hence, the conjunction of all nested annotations for a code chunk is usually referred as this code's *presence condition* [17, 70]. In our example, Listing 2.1, the presence condition of both code artifacts is equal to feature mapping because there is no nesting.

Different formalisms exist to model annotative variability [14, 21, 25, 29], one of which are variation trees [10]. A variation tree is a graph that represents the hierarchy between code blocks and their annotations. Figure 1 shows the variation tree corresponding to Listing 2.1. Each node either represents a feature mapping (blue outline) or a code artifact (black outline). The numbers in the nodes

```
#ifdef A
void f(){
  ...
}
#endif

#ifdef A || B
void g(){
  ...
}
#endif
```

Listing 2.1: Example of CPP annotations to implement variability

```
- #ifdef A
+ #ifdef A && B
void f(){
  ...
}
#endif

- #ifdef A || B
+ #ifdef B
void g(){
  ...
}
#endif
```

Listing 2.2: Patch that changes annotations in Listing 2.1

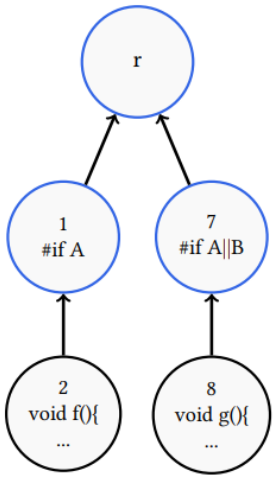

Figure 1: Variation Tree of Listing 2.1

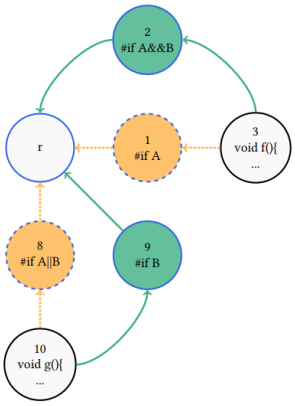

Figure 2: Variation Diff of Listing 2.2

reference their line number and edges describe the nesting hierarchy of annotations. In this example, the artifact node 2 with an edge pointing to a feature mapping node 1 means that the function `f` is directly annotated with the feature mapping formula *#if A*.

Definition 2.1 (Variation Tree [10]). A variation tree (V, E, r, τ, l) is a tree with nodes V, edges $E \subseteq V \times V$, and root node $r \in V$. Each edge $(x, y) \in E$ connects a child node x with its parent node y, denoted by $p(x) = y$. The node type $\tau(v) \in \{\texttt{artifact}, \texttt{mapping}, \texttt{else}\}$ identifies a node $v \in V$ either as representing an implementation artifact, a feature mapping, or an else branch. The label $l(v)$ is a propositional formula if $\tau(v) = \texttt{mapping}$, a reference to an implementation artifact if $\tau(v) = \texttt{artifact}$, or empty if $\tau(v) = \texttt{else}$. The root r has type $\tau(r) = \texttt{mapping}$ and label $l(r) = \mathit{true}$. An else node can only be placed directly below a non-root mapping node and a mapping node has at most one child of type else.

2.2 Edits to Software Product Lines

In software product lines, code changes may affect source code as well as feature mappings. As an example, Listing 2.2 shows a patch to Listing 2.1, which changes the feature mappings of both

functions f and g. Hence, both functions appear in different variants after the patch as compared to before the patch. While this is a toy example, more complex cases arise in practice because the C-preprocessor is used at a large-scale to implement variability [27, 40]; and sometimes excessive usage of annotations is referred to as the `ifdef`-hell [22, 38, 43, 47].

To understand how changes to feature mappings affect the source code, *variation diffs* have been introduced as a graph-based model to desribe changes to variation trees [10], and hence changes to feature mappings as well as source code. Variation diffs describe changes by coloring a graph to denote whether a node or edge was added, removed, or left unchanged. As an example, Figure 2 shows the corresponding variation diff for Listing 2.2, and hence embodies a patch ot Figure 1. Similar to the diff notation, dotted nodes and edges in orange were deleted during the respective edit, while green nodes and edges were added.

Definition 2.2 (Variation Diff [10]). A variation diff is a rooted directed connected acyclic graph $D = (V, E, r, \tau, l, \Delta)$ with nodes V, edges $E \subseteq V \times V$, root node $r \in V$, node types τ, and node labels l as defined for variation trees (Definition 2.1). The additional function $\Delta : V \cup E \rightarrow \{-, +, \bullet\}$ defines whether a node or edge was added $+$, removed $-$, or unchanged $\bullet$, such that $project(D, t)$ is a variation tree for all times $t \in \{before, after\}$.

A variation diff describes the superimposition of the two variation trees *before* and *after* the change. With the function $project(D, t)$, we can extract the variation tree from a variation diff D at time $t \in \{before, after\}$. Thereby, the predicate $exists(t, \Delta(x))$ is true iff the edge or node x exists at the given time t.

$$project((V, E, r, \tau, l, \Delta), t) := (\{v \in V \mid exists(t, \Delta(v))\}, \{e \in E \mid exists(t, \Delta(e))\}, r, \tau, l).$$

In the rest of this paper, we use the definitions of variation trees and variation diffs as a basis for our formalism and extend it to model relations between feature mappings other than the nesting hierarchy.

3 VARIANT SET RELATIONS

To gain insights into the evolution of feature mappings in evolving software product lines, we introduce a formalism to model relations between feature mappings changed during edits that enables automatic analyses. We first describe how we can relate feature mappings in a meaningful way, and then show how these relations can be detected.

3.1 Relations Between Feature Mappings

While a feature mapping identifies an artifact with the features it belongs to, the mapping essentially denotes a subset of the variants that contain that artifact. An artifact mapped to a feature or feature mapping formula X is present in at most all variants that implement that feature or combination of features X. In the remainder of this paper, we denote the set of all variants described by a feature mapping formula X as $[\![X]\!]$. For example, $[\![A \wedge B]\!]$ denotes the set of all variants with features A and B present.

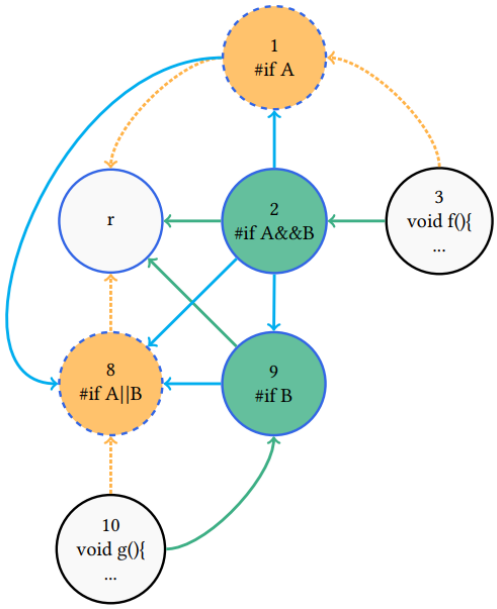

Figure 3: Edge-Typed Variation Diff with implication edges (cyan) for Listing 2.2

In general, there are three prevalent relations between two sets: (1) One set can be a subset of the other, (2) the two sets do not share any elements, or (3) the sets have some but not all elements in common. In turn, when two sets of variants relate in one of these ways, their respective feature mappings should relate as well. The observation is that these basic relations on set can be lifted to relations on propositional formulas that may denote feature mappings.

3.1.1 Subset. A set of variants $[\![X]\!]$ of feature mapping X is a subset of another set of variants $[\![Y]\!]$ of feature mapping Y *iff* $[\![X]\!] \subseteq [\![Y]\!]$. This means, all variants described by the feature mapping X are also described by the feature mapping Y, but the feature mappings Y may describe additional variants. Transferred to the feature mappings X and Y, this is equivalent to X implying Y for every variable assignment. We write "X implies Y for every variable assignment" as $X \models Y$ (reads as: X models Y). The expression $X \models Y$ is a tautology check $TAUT(X \Rightarrow Y)$ and can therefore be checked with a satisfiability solver in terms of $\neg SAT(\neg(X \Rightarrow Y))$.

3.1.2 Disjoint. Two sets of variants $[\![X]\!], [\![Y]\!]$ of feature mappings X, Y are disjoint *iff* $[\![X]\!] \cap [\![Y]\!] = \emptyset$. The respective feature mappings X and Y describe disjunct sets of variants. Since $[\![X]\!]$ and $[\![Y]\!]$ are disjoint sets, there can be no variant described by both formulas X and Y. Therefore, for every possible variable assignment, only one of X or Y or none of both can evaluate to *true*. From this, we can derive that once we know X is true, Y cannot be true, and vice versa. Expressing this mutual exclusion in the notation we introduced for implication edges leaves us with $X \models \neg Y$. In the context of product line configurations, this means that a selection of features can either satisfy formula X or Y, but never both. Two

code blocks, one directly annotated with X and one annotated with Y, can therefore never occur in the same variant.

3.1.3 Intersected. Two sets are intersected if they have some but not all elements in common. Therefore, two sets of variants $[\![X]\!]$, $[\![Y]\!]$ of feature mappings X, Y are intersected *iff* $[\![X]\!] \cap [\![Y]\!] \neq \emptyset$, $[\![X]\!] \setminus [\![Y]\!] \neq \emptyset$ and $[\![Y]\!] \setminus [\![X]\!] \neq \emptyset$. The code artifacts annotated with X and Y are independent:They can both be present in the same variants but also appear separately in different variants.

We derive three relations *Subset*, *Disjoint*, and *Independent* from the three presented relations between two sets of variants, respectively. These relations are defined between pairs of feature mapping nodes in variation diffs and variation trees. The formal definitions can be found in our appendix, which is hosted in our github repository[1]. We call the set of the three relations the *variant set relations* (*vset* relations). In the following, we show that the variant set relations form a complete set and how we can classify edited feature mappings within these three relations.

3.2 Completeness & Unambiguity

As known from set theory, and as visible from the formulas, the relations Subset, Disjoint, and Intersected form a complete classification of two sets: Every pair of sets is in at least one of these relations. Consequently, if we relate feature mappings in a variation tree or diff with edges representing these relations, also the sets of edges of each type in a variation tree are complete. This means that there exists an explanation for every pair of feature mappings (in a variation tree or diff).

Corollary 1 (Completeness of Variant set relations). *Given an arbitrary set of nodes V of a variation tree or diff, then for any two mapping nodes $x, y \in V, \tau(x) = \tau(y) = mapping$:*

$$(x, y) \in (Implication \cup Alternative \cup Independent).$$

An explicit proof of this corollary is part of our appendix[1].

While the *vset* relations are complete, they remain ambiguous. This means that for any pair of feature mappings, there might exist multiple explanations. In particular, when $l(x) \equiv l(y) \equiv False$ the two feature mapping nodes can be assigned to *Implication* as well as *Alternative*. We can get rid of this ambiguity by explicitly classifying this case as an *Alternative* since the source code nesting below the respective nodes cannot be present in the same variant (as they are not present in any variant) and explicitly exclude this case from the *Implication* relation. With this constraint, every pair of feature mapping nodes is in exactly one of the three relations, making the *vset* relations additionally unambiguous.

4 EXPLAINING EDITS TO FEATURE ANNOTATIONS

The goal of this paper is to help explain edits to feature mappings during the development of software product lines. In the previous Section 3, we observed that feature mappings can be compared based on their semantics $[\![.]\!]$ (i.e., the set of variants they denote). So given *that* we can relate feature mappings, we now turn to identify *when* and *which* feature mappings should be related upon an edit to a software product line. In particular, we extend variation trees and variation diffs to model and automatically compute relations, and thus enable explanations for edits.

4.1 Edge-Typed Variation Trees

To inspect how feature mappings are related, we choose a formalism that already models feature mappings explicitly. Variation trees [10] satisfy our requirements. To model *vset* relations in variation trees, we have to extend the model of variation trees to include new edges representing binary relations between the formulas of feature mapping nodes.

Therefore, we extend the set of edges E of a variation tree to not only include the ordinary edges, which model the nesting structure of annotations, but to also include edges between any two nodes that are in a relation $R \subseteq V \times V$. To identify the relation an edge represents, we introduce an edge typing π that maps edges to their respective relation:

$$\pi : E \rightarrow P(V \times V),$$

where P denotes the powerset function and E is a set of edges. Note that since the *vset* relations are defined as sets of edges, the *vset* relations are all elements of $P(V \times V)$. As an example, edges in a standard variation tree all represent the same relation: the nesting hierarchy of annotations. Hence, we type nesting edges $e \in E$ as $\pi(e) = Nesting$ where $e \in Nesting \subseteq V \times V$ is indirectly specified by the developers of the product line. Note that, while we are mainly interested in relations between feature mappings in this work, our extension in fact can model relations between any pair of nodes.

The definition of edge-typed variation trees is a generalization of standard variation trees:

Definition 4.1 (Edge-Typed Variation Tree). An edge-typed variation tree is a tuple (V, E, r, τ, l, π) with nodes V, edges $E \subseteq V \times V$, an edge label function $\pi : E \rightarrow P(V \times V)$ such that that $\forall e \in E : e \in \pi(e)$ and r, τ, l as defined in Definition 2.1 if and only if $reduce_{VT}((V, E, r, \tau, l, \Delta, \pi))$ is a variation tree.

An Edge-typed variation tree is valid if the non-nesting edges augment a variation tree. We can obtain the underlying variation tree by including only nesting edges and leaving out the other edges with $\pi(e) \neq Nesting$:

$$reduce_{VT}((V, E, r, \tau, l, \pi)) := (V, \{e \in E \mid \pi(e) = Nesting\}, r, \tau, l).$$

4.2 Edge-Typed Variation Diffs

To explain edits to feature mappings, we also want to use a formalism that already models feature mappings explicitly, as well as evolution of source code. Therefore, we do not have to either add a representation of variability information nor evolution to the formalism but rather only the relations between edited, already modeled, variability information. We therefore choose variation diffs [10] (cf. Subsection 2.2) as the basis for our formalism, which are sound and complete. Analogous to variation trees, the representation of feature mappings as their own nodes with the respective feature mapping formulas as the node labels allow us to introduce relations as additional edges, thereby retaining the graph structure.

[1] https://github.com/guethilu/DiffDetective/blob/vamos24_replication/appendix/VaMoS_Appendix.pdf

However, variation diffs do not yet relate feature mappings on their own, since per definition, feature mapping nodes only store their own variability information in their labels and no references to other nodes in the graph. Edges only represent the nesting structure of the source code and do not reflect other potential relations between nodes. For example, the feature mapping node 8 in Figure 2, that was Node `10`'s feature mapping before the edit, is implied by the new feature mapping, Node 9. Therefore, the code referenced by the code artifact node `10` appears in a narrower set of variants after the edit than before.

Formally, we define *edge-typed variation diff* as an extension variation diffs that also include an edge typing π:

Definition 4.2 (Edge-Typed Variation Diff). An edge-typed variation diff is a graph $(V, E, r, \tau, l, \Delta, \pi)$ with nodes V, edges $E \subseteq V \times V$ and an edge label function π as defined in Definition 4.1, and r, τ, l, Δ as defined in Definition 2.2 if and only if $reduce_{VD}((V, E, r, \tau, l, \Delta, \pi))$ is a variation diff.

Analogous to edge-typed variation trees, an edge-typed variation diff is valid if its reduction yields a valid variation diff. Edge-typed variation diffs can be reduced to ordinary variation diffs by including only nesting edges (i.e., edges $e \in E$ with $\pi(e) = Nesting$) and discarding the typing π:

$$reduce_{VD}((V, E, r, \tau, l, \Delta, \pi)) := (V, \{e \in E \mid \pi(e) = Nesting\}, r, \tau, l, \Delta).$$

We need this correctness criterion to ensure that the underlying hierarchy of the source code is still represented, no matter what relations are added to the variation diff as additional edges.

The reduced edge-typed variation diffs can then be projected to variation trees with the *project* function as described in Subsection 2.2. Analogously, edge-typed variation diffs can also be projected to edge-typed variation trees. For the projection of edge-typed variation diffs to edge-typed variation trees, we define a function $project_{ET}$ that ensures that edges remain in the tree only when both their nodes exist at the projected time. For example, an edge between a node that was deleted and a node that was added can neither appear in the corresponding edge-typed variation tree before the edit nor after because one of the respective nodes does not exist in the graph. Therefore, we define

$$project_{ET}((V, E, r, \tau, l, \Delta, \pi), t) := (\{v \in V \mid exists(t, \Delta(v))\}, E', r, \tau, l, \pi).$$

where

$$E' := \{e \in E \mid \pi(e) = Nesting \wedge exists(t, \Delta(e))\} \cup \{(x, y) \in E \mid \pi((x, y)) \neq Nesting \wedge exists(t, \Delta(x)) \wedge exists(t, \Delta(y))\}$$

This $project_{ET}$ is similar to the *project* for ordinary variation diffs and variation trees, however, it additionally checks if a relation edge connects two nodes that are present at the same point of time.

The definitions of the *reduce* functions together with the *project* functions for edge-typed variation diffs as well as for ordinary variation diffs leads us to the commuting square depicted in Figure 4.

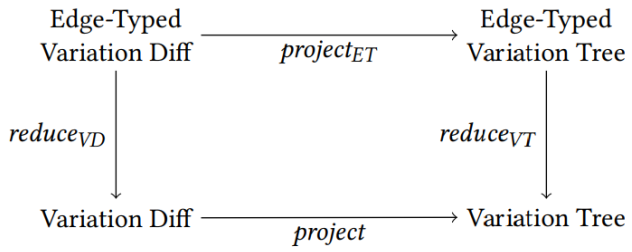

Figure 4: Commuting square to visualize possible transformations from edge-typed variation diffs to edge-typed variation trees and variation diffs

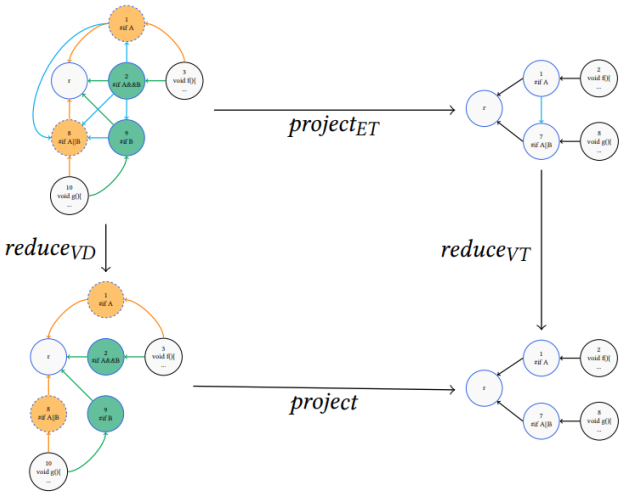

Figure 5: Commuting square for our running example

Note that the soundness of edge-typed variation diffs is ensured by the commutativity of projection and reduction, i.e., that reducing an edge-typed variation diff and projecting the resulting variation diff to either after or before an edit leaves us with the same variation tree as first projecting the same edge-typed variation diff to an edge-typed variation tree and the reducing that edge-typed variation tree to a variation tree.

Lemma 4.3. *For any edge-typed variation diff D and any time $t \in \{\text{before}, \text{after}\}$ reduction and projection commute:*

$$reduce_{VT}(project_{ET}(D, t)) = project(reduce_{VD}(D), t).$$

We prove this lemma in our appendix[1]. Our running example's reduction-projection square is illustrated in Figure 5.

4.3 Implementation

To use edge-typed variation diffs in practice (e.g., for CI/CD analysis tooling), their construction should be automatable. For this, we implemented an extension to `DiffDetective` with an algorithm that turns a variation diff into an edge-typed variation diff. This algorithm iterates over all pairs of feature mapping nodes and checks if they are in the *Implication* or *Alternative* relation. We

optimized the construction by making use of the mutual exclusivity of *Implication* and *Alternative* to save SAT checks (cf. Corollary 1). Once we find out that a pair of feature mapping nodes is in the *Alternative* relation, we do not have to check for the *Implication* relation. We check for *Alternative* first since it is symmetric, so one SAT check suffices, whereas we have to check *Implication* in both directions. Our implementation is hosted on GitHub[2].

5 FEASIBILITY STUDY

In this section, we analyze the relations of edited feature annotations in real-world software product lines and thereby investigate the feasibility of detecting such relations in practice. In particular, we evaluate our formalism regarding construction performance of edge-typed variation diffs as well as occurrence of the *vset* relation edges in real-world version histories. This evaluation serves as an initial study on the feasibility and potential impact of edge-typed variation diffs in potential tools for software product-line analyses.

Our first research question aims to assess the feasibility of identifying relations of feature annotations for software product-line analysis tools. For instance, edge-typed variation diffs could be used in automated analyses of changed variability information, such as a continuous integration pipeline, if their construction is reasonably fast.

RQ1: What is the runtime for identifying relations of feature annotation?

Second, we would like to assess the potential impact of detecting whether edited feature annotations are related in practice. If *implication* or *alternative* edges occur frequently in edge-typed variation diffs, analysis tools could make heavy use of this information. Otherwise, when edited annotations are in no relation (i.e., in the *independent* relation) variability would experience more extensive changes in practice that may alter products in an unrelated way.

RQ2: How many edge-typed variation diffs constructed from real software contain *Implication* or *Alternative* edges?

With our last research question, we inspect how often each relation occurs in real-world edits. This question gives us insight into how often edits by developers lead to related annotations and in which way.

RQ3: How often are edited annotations related in terms of *implication* or being *alternative*?

5.1 Experiment Setup

To answer our research questions, we analyze the version histories of real-world and open-source software product lines. Therefore, we implement edge-typed variation diffs as an extension into `DiffDetective` [10, 68], a framework for analyzing git histories of C preprocessor-based product lines. `DiffDetective` parses all patches in all commits in a repository's history to a variation diff per patch. Thus each parsed variation diff describes all changes from a certain commit to a particular file.

For each variation diff parsed by `DiffDetective`, we apply our algorithm to turn that diff into an edge-typed variation diff, and consequently obtain the relations of all changed feature annotations throughout all patches in the history. We measure the runtime necessary for the algorithm to run (RQ1). We then count the number of edge-typed variation diffs that contained at least one new edge (RQ2), as well as the number of (feature mapping) nodes in the variation diff to gain an insight into how much the variability information *can* change at all. Additionally, we count the added implication and alternative edges (RQ3).

Our test system is a Lenovo Thinkpad running Windows 11 with an Intel i7-1260P processor and 32 GB of RAM. During the experiment, we did not run other non-operating system software in parallel to reduce potential deviation in runtime measurements.

[2]https://doi.org/10.5281/zenodo.10276785

5.2 Subject Systems

We run our experiment on 42 real-world open-source software product lines that implement variability through C preprocessor annotations. The subject systems are a subset of the subject systems used by Bittner et al. [10]. We also use Busybox, Marlin, libssh, Godot, and 38 of the 40 systems used by Liebig et al. [41]. We omitted two repositories due to compatibility reasons since we ran our evaluation on a Windows system and the respective repositories contained folders with names forbidden under Windows. This selection of software systems covers various domains, ranging from operating systems and printer firmware to text editors and game engines.

5.3 Results and Discussion

With the 42 repositories and all 862,969 combined commits in their respective commit history, we analysed a total of 3,117,797 patches. In the following, we present both the mean and the median values for runtime and numbers of edges, since the outliers heavily influence the arithmetic means. We think that both metrics still have their raison d'être, since the mean values can act as an indication for potential usage in tools, while the median values give an intuition for the vast majority of commits and patches.

RQ1: Runtime. To answer this question, we measured the time needed to build a variation diff and extend it to an edge-typed variation diff. On average, extending variation diffs to edge-typed variation diffs took a mean of 37.50 ms per variation diff. Grouped to commits, this results in an average runtime of 135.49 ms per commit. The number of patches and commits grouped by their respective runtimes is shown in Table 1. However, the median runtime for both patches and commits is less than 1 ms, as 90.50% of patches and 83.48% of commits are below the measurement threshold. This low median runtime can be explained by the results of the next research question. Both median and mean imply that edge-typed variation diffs can be a feasible formalism to be used in tools (e.g., in a pipeline in version control) because the runtime is low. Hence, we hypothesize that in a scenario where edge-typed variation diffs are employed for analyses, the main computational load likely comes from the actual analysis run on each edge-typed variation diff, not from their construction.

RQ2: #Edits with Derivable Relations. To answer **RQ2**, we counted the number of patches in version histories for which *implication* or *alternative* relation edges exist in the corresponding edge-typed variation diff. Of all 3,117,797 edge-typed variation diffs, 182,966 contain relation edges, which is about 6% of patches. The exact number of patches with and without relation edges is shown

	<0 ms		[1 ms,10 ms]		(10 ms, 100 ms]		(100 ms, 1 s]		(1 s, 1 min]		>1 min		∑
	#	%	#	%	#	%	#	%	#	%	#	%	#
Commits	720,439	83.48	90,423	10.48	37,587	4.36	10,845	1.26	3,460	0.40	215	0.02	862,969
Patches	2,821,529	90.50	199,471	6.40	73,531	2.36	18,334	0.59	4,750	0.15	182	0.01	3,117,797

Table 1: Share of patches and commits with runtimes in the specified time intervals

No relation edges		*Implication*		*Alternative*		*Implication* and *Alternative*		∑
#	%	#	%	#	%	#	%	#
2,941,302	94.34	176,495	5.67	26,358	0.85	19,887	0.64	3,117,797

Table 2: Number of patches with and without *vset* relation edges.

		All Variation Diffs		Changed Variation Diffs	
	Total	Mean	Median	Mean	Median
Implication	62,522,349	20.05	0	354.24	4
Alternative	2,011,486	0.65	0	76.31	4

Table 3: Number of *Implication* and *Alternative* edges added to variation diffs

in Table 2. About 85.5% of edge-typed variation diffs that have at least one relation edge, only have implication edges. About 3.5% of these patches have only alternative edges. About 11% of these patches contain both edge types. Moreover, we found that the number of feature mappings nodes in the inspected variation diffs is 2 in median. Given that one of the two nodes is the mandatory root node, this low median could explain the low runtimes reported for **RQ1**: The root node has *true* as its feature mapping formula, however, we ignore this node in our edge-adding algorithm since the resulting relation edge would not represent meaningful information as the edges between feature mapping nodes do. We can not add an implication or alternative edge in a variation diff when only one actual feature mapping node is present. In our implementation, the iteration of feature mappings nodes does not even start and the procedure to add edges terminates immediately.

RQ3: #Implication Edges & #Alternative Edges. To answer our last research question, we counted the occurrences of implication and alternative relation edges in all edge-typed variation diffs. We identified a total of about 62 million implication edges, with an average of 20.05 implication edges per edge-typed variation diff. Due to most edge-typed variation diffs not having any relation edges, the median is zero. If we disregard diffs without any relation edge, we get an average of 354.24 implication edges per edge-typed variation diff, and a median of 4 implication edges.

In total, about 2 million alternative edges have been added to edge-typed variation diffs, with an average of 0.65 and a median of 0 per edge-typed variation diff. Disregarding diffs without relation edges, edge-typed variation diffs on average had 76.31 alternative edges, and 4 in median.

The high number of overall relation edges together with the low number of patches that actually contain relation edges implies that there is a vast majority of edits with little to no variability and a minority of edits with a lot of highly interconnected variability information.

The mean values are heavily affected by outliers. For example, the commit with the longest runtime (about 3 h) was commit `0181d0...` [3] from the game engine Godot, in which the shader's SDK got updated, which resulted in a commit with 74 changed files with 104,239 additions and 85,420 deletions. The same commit also reported almost 8 million implication edges due to many similar *#if* and *#ifdef* directives in the diff of one file in the shader's source code.[4]

5.4 Threats to Validity

Internal Validity. We rely on the correctness of `DiffDetective` to load Git repositories, extract commits and patches and build variation diffs. Although Bittner et al. claim to have tested all crucial functionality, however, there might be bugs in the original implementation as well as in the additions we made to `DiffDetective` that could lead to wrong results. For our extensions, we also tested our edge-adding algorithm and manually spot checked the results.

For measuring the time necessary to extend the variation diff of a single patch to an edge-typed variation diff, we chose Java's `System.currentTimeMillis()`, reporting a runtime of zero milliseconds for ~91% of all patches. A finer granularity in the runtime measurement could be implemented in our experiment setup. Yet, regarding feasibility and most analyses, runtimes below this measurable threshold are negligible such that a finer time measurement would not change the interpretation of our results.

External Validity. All our datasets are C/C++ software projects that implement variability using the C preprocessor which could limit the generalizability of our results. We had to omit two repositories from the datasets analyzed by Bittner et al. [10] due to compatibility issues. Yet, the datasets in this study cover a wide variety of different domains. Moreover, the C preprocessor is still one of the most widely used mechanisms to implement static variability in practice.

[3] https://github.com/godotengine/godot/commit/0181d005c98dcf62fd63e19b0dec7586ac708154

[4] https://github.com/godotengine/godot/commit/0181d005c98dcf62fd63e19b0dec7586ac708154#diff-7b55b2dc0d13f1e7392fc0cd2e6cfc9046f365f0850226b170aa010738eb14c4

6 RELATED WORK

As mentioned in Section 4, our work mainly extends the work of Bittner et al. [10]. In this section, we compare our work to previous research on the evolution of software product lines as well as potential applications of edge-typed variation diffs in variation control systems or incremental analyses.

6.1 Variability and Its Evolution

After inspecting variability in 40 preprocessor-based software product lines, Liebig et al. [41] conclude that most of the source code of configurable software is not variational, i.e., on average 77% of the source code is code without feature annotations. This correlates with our findings that for most edits the complexity of edge-typed variation diffs does not change, since for edits on that 77% of source code, there are no feature mappings to be compared and therefore no relation edges to be added.

Stănciulescu et al. [63] categorize edits to source code, both covering functionality as well as variability, using the choice calculus [21, 71, 72]. They propose several patterns of edits to variational source code, sorting them into *Code-Adding Patterns*, *Code-Removing Patterns*, and *Other Patterns*. The evolution of feature mappings itself is only partly covered with their pattern *ChangePC*, which describes unchanged code with a changed feature annotation and a thereby changed presence condition. The pattern *ChangePC*, however, does not give any information about the relationship between the removed and the added feature annotation. Additionally, Bittner et al. [10] show that Stănciulescu et al.'s proposed patterns lack completeness and unambiguity regarding edit operations on source code in configurable software systems.

The choice calculus [21, 71, 72] is a formalism to describe software variability. While the choice calculus has been applied successfully for different problems [7, 63, 72, 73] and can be used to describe variability in time (i.e., changes to software), or variation in space (i.e., the similarities and differences between software variants), it – to the best of our knowledge – has not been used to combine both spaces simultaneously to describe evolution of variable software yet.

In older works on software evolution, software product lines or feature evolution are typically not addressed. Chapin et al. [15] avoid the term feature in their work about the different types of software evolution due to a lack of an unambiguous definition of feature, while Hsi and Potts [26] write about the evolution and enhancement of software features but without any connection to configurable software. This lack of consensus about the terminology is also reported by Marques Samary et al. [45], who did a literature review on different approaches to support software product line evolution.

Our explanations of edits to variability annotations in terms of variant set relations are quite similar to those introduced to explain feature model evolution [12, 67]. The similarity arises because for both, variability annotations and feature models, sets of variants are compared in terms of tautology checks on propositional formulas. While feature models are essentially propositional formulas [8] that can be compared directly, we additionally have to determine *which* variability annotations should be compared (cf. Section 4) as there are multiple, scattered annotations per evolution step.

6.2 Applications and Use-Cases

Linsbauer et al. [42] give a categorization of variation control systems. The information on the evolution of feature annotations represented in edge-typed variation diffs can help with some of the problems Linsbauer et al. cite as the reasons why variation control systems are not used in practice today. For example, the cognitive complexity that comes with complex logical expressions could be reduced when developers do not have to look at two separate formulas but rather two formulas in a prior known relation.

Similarly, more recent projects that aim to enable automatized synchronization of different software variants, for example in managed clone-and-own [9, 31, 44, 52, 56, 58, 59, 61], could make use of edge-typed variation diffs. The approach of using a session-oriented edit model already establishes proximity to variation diffs, since an edit made in one session can be represented as, for example, a git commit, which then can be parsed to an (edge-typed) variation diff. Just as using abstract syntax trees for propagating changes to other variants, using edge-typed variation diffs could be a feasible approach. Edge-typed variation diffs could also be inspected when bringing changes on feature models and changes on feature mappings in relation, like Seidl et al. did [62].

Similarly, analysis tools using mutation operators [2] could be adapted with edge-typed variation diffs, since for example Al-Hajjaji et al.'s operators *RFIC* or *RIND* directly relate to our implication edges (*RFIC*) or alternative edges (*RIND*).

The information derivable from edge-typed variation diffs could also be used when restructuring or refactoring software product lines. Laguna and Crespo [37] compare different refactorings that can be done to software product lines, along with the intentions and the tools used for the respective refactoring. For example, a tool recommending refactorings proposed by Ribeiro and Borba [57] could use the information from edge-typed variation diffs to adapt recommendations for the respective preprocessor-based files. Furthermore, tools for developing software product lines like FeatureIDE [39, 48], CIDE [36], or Colligens [46] could extend their functionality regarding evolution of source code with edge-typed variation diffs.

7 CONCLUSION

In this work, we inspect the evolution of feature annotations by introducing edge-typed variation diffs, variation diffs [10] extended with additional edges. To model relations between edited feature mappings, we extend variation diffs with a set of edges representing relations between feature mapping nodes in the corresponding variation diffs. These variant set relations consist of *Implication*, *Alternative*, and *Independent* edges, where every relation represents the relation between the sets of variants described by two feature mappings. We further show that the set of variant set relations is complete, i.e., every pair of feature mapping nodes in an edge-typed variation diff is in exactly one of the mentioned relations.

To conduct experiments on real-world source code, we extended `DiffDetective` to build edge-typed variation diffs from software product-line repositories. In our experiment, we found that most edits made to source code do not change the variability hierarchy of the respective source code. However, negligible runtime overhead comes with extending variation diffs to edge-typed variation diffs,

as even for the ~6% of edge-typed variation diffs that changed in complexity, the mean runtime was 135 ms. In these cases, a median of four implication and alternative edges each was added to the respective variation diffs, showing that edited variability information is usually closely related. Our edge-typed variation diffs help to detect and explain changes to feature mappings in terms of the sets of variants described by the respective feature mappings.

In the future, more relation types could be introduced to model more detailed relationships between feature mappings as well as code artifacts in the hierarchy of source code of variable systems. Based on our formal foundations, explanations of edits to variability annotations could be put to action in terms of user-centered applications to help developers of software product lines to better understand their changes. Furthermore, we plan to apply graph mining techniques like frequent sub-graph mining to edge-typed variation diffs in order to find frequent patterns occurring during the development of configurable software.

ACKNOWLEDGMENTS

We thank our reviewers for their constructive feedback. We also thank Christof Tinnes, Timo Kehrer, and Alexander Schultheiß for helpful discussions. This work has been partially funded by the German Research Foundation within the projects *VariantSync* (TH 2387/1-1) and *Co-InCyTe* (SCHA1635/15-1 and LO 2198/4-1).

REFERENCES

[1] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *Trans. on Software Engineering and Methodology (TOSEM)* 26, 3, Article 10 (2018), 10:1–10:34 pages.

[2] Mustafa Al-Hajjaji, Fabian Benduhn, Thomas Thüm, Thomas Leich, and Gunter Saake. 2016. Mutation Operators for Preprocessor-Based Variability. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 81–88.

[3] Vander Alves, Pedro Jr, Leonardo Cole, Paulo Borba, and Geber Ramalho. 2005. Extracting and Evolving Mobile Games Product Lines. 70–81. https://doi.org/10.1007/11554844_8

[4] Sven Apel. 2007. *The Role of Features and Aspects in Software Development*. Dissertation. University of Magdeburg, Germany.

[5] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[6] Sven Apel, Christian Kästner, and Christian Lengauer. 2013. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Trans. on Software Engineering (TSE)* 39, 1 (2013), 63–79.

[7] Parisa Ataei, Fariba Khan, and Eric Walkingshaw. 2021. A Variational Database Management System. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 29–42.

[8] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 7–20.

[9] Paul Maximilian Bittner, Alexander Schultheiß, Thomas Thüm, Timo Kehrer, Jeffrey M. Young, and Lukas Linsbauer. 2021. Feature Trace Recording. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 1007–1020.

[10] Paul Maximilian Bittner, Christof Tinnes, Alexander Schultheiß, Sören Viegener, Timo Kehrer, and Thomas Thüm. 2022. Classifying Edits to Variability in Source Code. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 196–208.

[11] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. 2012. A Theory of Software Product Line Refinement. *Theoretical Computer Science* 455, 0 (2012), 2–30.

[12] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. 2015. Reasoning About Product-Line Evolution Using Complex Feature Model Differences. *Automated Software Engineering* 23, 4 (2015), 687–733.

[13] Rafael Capilla, Jan Bosch, Kyo-Chul Kang, et al. 2013. Systems and software variability management. *Concepts Tools and Experiences* 10 (2013), 2517766.

[14] Thiago Castro, Leopoldo Teixeira, Vander Alves, Sven Apel, Maxime Cordy, and Rohit Gheyi. 2021. A Formal Framework of Software Product Line Analyses. *Trans. on Software Engineering and Methodology (TOSEM)* 30, 3, Article 34 (2021), 37 pages.

[15] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, and Wui-Gee Tan. 2001. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice* 13, 1 (2001), 3–30. https://doi.org/10.1002/smr.220 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.220

[16] Fanny Chevalier, David Auber, and Alexandru Telea. 2007. Structural Analysis and Visualization of C++ Code Evolution Using Syntax Trees. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting* (Dubrovnik, Croatia) *(IWPSE '07)*. Association for Computing Machinery, New York, NY, USA, 90–97. https://doi.org/10.1145/1294948.1294971

[17] Krzysztof Czarnecki and Michal Antkiewicz. 2005. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. Springer, 422–437.

[18] Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley.

[19] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2018. FEVER: An Approach to Analyze Feature-Oriented Changes and Artefact Co-Evolution in Highly Configurable Systems. *Empirical Software Engineering (EMSE)* 23, 2 (2018), 905–952.

[20] M.D. Ernst, G.J. Badros, and D. Notkin. 2002. An empirical analysis of c preprocessor use. *IEEE Transactions on Software Engineering* 28, 12 (2002), 1146–1170. https://doi.org/10.1109/TSE.2002.1158288

[21] Martin Erwig and Eric Walkingshaw. 2011. The Choice Calculus: A Representation for Software Variation. *Trans. on Software Engineering and Methodology (TOSEM)* 21, 1, Article 6 (2011), 27 pages.

[22] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. 2013. Do Background Colors Improve Program Comprehension in the #Ifdef Hell? *Empirical Software Engineering (EMSE)* 18, 4 (2013), 699–745.

[23] Felype Ferreira, Paulo Borba, Gustavo Soares, and Rohit Gheyi. 2012. Making Software Product Line Evolution Safer. In *2012 Sixth Brazilian Symposium on Software Components, Architectures and Reuse*. 21–30. https://doi.org/10.1109/SBCARS.2012.18

[24] Felype Ferreira, Rohit Gheyi, Paulo Borba, and Gustavo Soares. 2014. A Toolset for Checking SPL Refinements. *J. Universal Computer Science (J.UCS)* 20, 5 (2014), 587–614.

[25] Alexander Gruler. 2010. *A Formal Approach to Software Product Families*. Ph. D. Dissertation. TU München.

[26] Hsi and Potts. 2000. Studying the evolution and enhancement of software features. In *Proceedings 2000 International Conference on Software Maintenance*. 143–151. https://doi.org/10.1109/ICSM.2000.883033

[27] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. 2016. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering (EMSE)* 21, 2 (2016), 449–482.

[28] Ralph E. Johnson and Brian Foote. 1988. Designing Reusable Classes. *J. of Object-Oriented Programming (JOOP)* 1, 2 (1988), 22–35.

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

[30] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. 2012. Type Checking Annotation-Based Product Lines. *Trans. on Software Engineering and Methodology (TOSEM)* 21, 3 (2012), 14:1–14:39.

[31] Timo Kehrer, Thomas Thüm, Alexander Schultheiß, and Paul Maximilian Bittner. 2021. Bridging the Gap Between Clone-and-Own and Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 21–25.

[32] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-Oriented Programming. In *Proc. Europ. Conf. on Object-Oriented Programming (ECOOP)*. Springer, 220–242.

[33] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. 2005. A case study in refactoring a legacy component for reuse in a product line. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. 369–378. https://doi.org/10.1109/ICSM.2005.5

[34] Christian Kröher, Lea Gerling, and Klaus Schmid. 2018. Identifying the Intensity of Variability Changes in Software Product Line Evolution. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 54–64.

[35] Christian Kröher, Lea Gerling, and Klaus Schmid. 2023. Comparing the Intensity of Variability Changes in Software Product Line Evolution. *J. Systems and Software (JSS)* 203 (2023), 111737.

[36] Christian Kästner. 2007. CIDE: Decomposing Legacy Applications into Features. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings. Second Volume (Workshops)*. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan, 149–150.

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

[38] Duc Le, Eric Walkingshaw, and Martin Erwig. 2011. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. In *Proc. Int'l Symposium on Visual*

Languages and Human-Centric Computing (VL/HCC). IEEE, 143–150.
[39] Thomas Leich, Sven Apel, Laura Marnitz, and Gunter Saake. 2005. Tool Support for Feature-Oriented Software Development - FeatureIDE: An Eclipse-Based Approach. In *Proc. Workshop on Eclipse Technology eXchange (ETX)*. ACM, 55–59.
[40] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 105–114.
[41] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 105–114. https://doi.org/10.1145/1806799.1806819
[42] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 49–62.
[43] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. 2006. A Quantitative Analysis of Aspects in the eCos Kernel. 40, 4 (2006), 191–204.
[44] Wardah Mahmood, Daniel Strueber, Thorsten Berger, Ralf Laemmel, and Mukelabai Mukelabai. 2021. Seamless Variability Management With the Virtual Platform. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 1658–1670.
[45] Maíra Marques Samary, Jocelyn Simmonds, Pedro Rossel, and M. Bastarrica. 2019. Software Product Line Evolution: a Systematic Literature Review. *Information and Software Technology* 105 (01 2019), 190–208. https://doi.org/10.1016/j.infsof.2018.08.014
[46] Flávio Medeiros, Thiago Lima, Francisco Dalton, Márcio Ribeiro, Rohit Gheyi, and Baldoino Fonseca. 2013. Colligens: A Tool to Support the Development of Preprocessor-Based Software Product Lines in C. In *Proc. Brazilian Conf: Software: Theory and Practice (CBSoft)*. CBSOFT.
[47] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. 2018. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Trans. on Software Engineering (TSE)* 44, 5 (2018), 453–469.
[48] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.
[49] Munge Development Team. 2011. Munge: A Purposely-Simple Java Preprocessor. Website. Available online at http://github.com/sonatype/munge-maven-plugin; visited on January 11th, 2011..
[50] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. 2005. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. *SIGSOFT Softw. Eng. Notes* 30, 4 (may 2005), 1–5. https://doi.org/10.1145/1082983.1083143
[51] Laís Neves, Paulo Borba, Vander Alves, Lucinéia Turnes, Leopoldo Teixeira, Demóstenes Sena, and Uirá Kulesza. 2015. Safe Evolution Templates for Software Product Lines. *J. Systems and Software (JSS)* 106 (2015), 42–58.
[52] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. 2012. Clone Management for Evolving Software. *IEEE Trans. on Software Engineering (TSE)* 38, 5 (2012), 1008–1026.
[53] Jörg Pleumann, Omry Yadan, and Erik Wetterberg. 2011. Antenna: An Ant-to-End Solution For Wireless Java. Website. Available online at http://antenna.sourceforge.net/; visited on November 22nd, 2011..
[54] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.
[55] Klaus Pohl and Andreas Metzger. 2018. *Software Product Lines*. 185–201. https://doi.org/10.1007/978-3-319-73897-0_11
[56] Daniela Rabiser, Paul Grünbacher, Herbert Prähofer, and Florian Angerer. 2016. A Prototype-Based Approach for Managing Clones in Clone-and-Own Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 35–44.
[57] Márcio Ribeiro and Paulo Borba. 2008. Recommending Refactorings When Restructuring Variabilities in Software Product Lines. In *Proceedings of the 2nd Workshop on Refactoring Tools* (Nashville, Tennessee) *(WRT '08)*. Association for Computing Machinery, New York, NY, USA, Article 8, 4 pages. https://doi.org/10.1145/1636642.1636650
[58] Julia Rubin and Marsha Chechik. 2013. A Framework for Managing Cloned Product Variants. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 1233–1236.
[59] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2013. Managing Cloned Variants: A Framework and Experience. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 101–110.
[60] Gabriela Sampaio, Paulo Borba, and Leopoldo Teixeira. 2019. Partially Safe Evolution of Software Product Lines. *J. Systems and Software (JSS)* 155 (2019), 17–42.
[61] Alexander Schultheiß, Paul Maximilian Bittner, Thomas Thüm, and Timo Kehrer. 2022. Quantifying the Potential to Automate the Synchronization of Variants in Clone-and-Own. In *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 269–280.
[62] Christoph Seidl, Florian Heidenreich, and Uwe Aßmann. 2012. Co-Evolution of Models and Feature Mapping in Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 76–85.
[63] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. 2016. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 323–333.
[64] Chico Sundermann, Michael Nieke, Paul Maximilian Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2021. Applications of #SAT Solvers on Feature Models. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 12, 10 pages.
[65] Alexandru Telea and David Auber. 2008. Code Flows: Visualizing Structural Evolution of Source Code. *Computer Graphics Forum* 27, 3 (2008), 831–838. https://doi.org/10.1111/j.1467-8659.2008.01214.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2008.01214.x
[66] Stephen W. Thomas, Bram Adams, Ahmed E. Hassan, and Dorothea Blostein. 2011. Modeling the Evolution of Topics in Source Code Histories. In *Proceedings of the 8th Working Conference on Mining Software Repositories* (Waikiki, Honolulu, HI, USA) *(MSR '11)*. Association for Computing Machinery, New York, NY, USA, 173–182. https://doi.org/10.1145/1985441.1985467
[67] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning About Edits to Feature Models. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 254–264.
[68] Sören Viegener. 2021. *Empirical Evaluation of Feature Trace Recording on the Edit History of Marlin*. Bachelor's Thesis. University of Ulm.
[69] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. 2005. CVSscan: Visualization of Code Evolution. In *Proceedings of the 2005 ACM Symposium on Software Visualization* (St. Louis, Missouri) *(SoftVis '05)*. Association for Computing Machinery, New York, NY, USA, 47–56. https://doi.org/10.1145/1056018.1056025
[70] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-Condition Simplification in Highly Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 178–188.
[71] Eric Walkingshaw. 2013. *The Choice Calculus: A Formal Language of Variation*. Ph. D. Dissertation. Oregon State University.
[72] Eric Walkingshaw and Klaus Ostermann. 2014. Projectional Editing of Variational Software. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 29–38.
[73] Jeffrey M. Young, Paul Maximilian Bittner, Eric Walkingshaw, and Thomas Thüm. 2022. Variational Satisfiability Solving: Efficiently Solving Lots of Related SAT Problems. *Empirical Software Engineering (EMSE)* 28 (2022).