Classifying Edits to Variability in Source Code

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

Christof Tinnes
christof.tinnes@siemens.com
Siemens AG
München, Germany

Alexander Schultheiß
alexander.schultheiss@hu-berlin.de
Humboldt University of Berlin
Berlin, Germany

Sören Viegener
soeren.viegener@uni-ulm.de
University of Ulm
Ulm, Germany

Timo Kehrer
timo.kehrer@inf.unibe.ch
University of Bern
Bern, Switzerland

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

ABSTRACT

For highly configurable software systems, such as the Linux kernel, maintaining and evolving variability information along changes to source code poses a major challenge. While source code itself may be edited, also feature-to-code mappings may be introduced, removed, or changed. In practice, such edits are often conducted ad-hoc and without proper documentation. To support the maintenance and evolution of variability, it is desirable to understand the impact of each edit on the variability. We propose the first complete and unambiguous classification of edits to variability in source code by means of a catalog of edit classes. This catalog is based on a scheme that can be used to build classifications that are complete and unambiguous by construction. To this end, we introduce a complete and sound model for edits to variability. In about 21.5 ms per commit, we validate the correctness and suitability of our classification by classifying each edit in 1.7 million commits in the change histories of 44 open-source software systems automatically. We are able to classify all edits with syntactically correct feature-to-code mappings and find that all our edit classes occur in practice.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; *Software evolution.*

KEYWORDS

software evolution, software variability, feature traceability, software product lines, mining version histories

ACM Reference Format:
Paul Maximilian Bittner, Christof Tinnes, Alexander Schultheiß, Sören Viegener, Timo Kehrer, and Thomas Thüm. 2022. Classifying Edits to Variability in Source Code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549108

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3549108

1 INTRODUCTION

In configurable software systems, such as the Linux kernel, certain code should only be present in certain *variants* of the software. For instance, parts of the code base may be platform dependent or a feature should only be available to a subset of customers. Maintaining and evolving variability information along changes to source code poses a major challenge for developers [77, 78, 89]. One aspect thereof is keeping track of changes to variable parts of the code base that implement different *features* or feature interactions of the configurable software. While source code itself may be edited, *feature-to-code mappings* may also be introduced, removed, or changed.

Awareness of edits to variability is crucial in the development of configurable software. For instance, edits to software product lines might introduce type errors in certain variants [40] or alter the set of available variants in an unintended way [69, 71, 84, 91]. In clone-and-own development, where each variant of a software is developed as a separate copy of the software (e.g., using branching or forking), changes to variants have to be tracked to update other variants accordingly [45, 50, 51, 53, 102]. Variation control systems [59, 60, 90] and managed clone-and-own methods [45, 63] inspect edits paired with information on edited features [10, 37] to recover knowledge about variability incrementally [61] or to reintegrate edits to a hidden unified code base [90].

In practice, however, the variability of the code base is often edited ad-hoc and without proper documentation. While changes are technically captured in terms of commits to a version control system, their impact on the variability is not explicitly accounted for and mostly opaque to developers and tools [34]. Instead of dealing with purely syntactical changes on the granularity level of commits, it is desirable to describe the difference between two versions of the code base as edits for which the effect on the variability is known. For example, one important class of edits to variability are refactorings, which change the structure but not the semantics of variability information. Knowing such effects is useful as further maintenance and evolution tasks can be simplified [11, 70, 71], such as adapting the test suite of a software product line [1, 62] or the propagation of changes between cloned variants or forks [45, 102].

Although the literature has recognized the need for characterizing variability evolution, a complete, unambiguous, and automated classification of edits does not yet exist. Studies on software product lines focus on specific scenarios such as variability-aware mutation testing [2] or safe evolution [69, 87, 88], or cover only edits to variability models [3, 95, 95] or configurations of variants [71].

Classifications of co-evolution of variability models and source code make no claims on completeness (i.e., there may be edits that cannot be classified) and require a manual identification of edits [11, 78, 89]. Similarly, classifications of edits in managed clone-and-own either require a manual edit investigation [37], or suffer from ambiguity [90], and incompleteness [37, 90].

To this end, we present a complete, unambiguous, and automatic classification of edits to variability in source code. We first introduce *variation trees* as a formalization for variability in source code. We then introduce *variation diffs* as a formalization for edits to variation trees, thus describing edits to variability in source code. We prove that variation diffs are complete and sound regarding variation trees, meaning that any possible edit to a variation tree is described by a variation diff and that every variation diff represents an actual edit to variation trees. By classifying all structures within variation diffs, we are able to classify all edits to variability in source code. We present a set of edit classes which we prove to be complete and unambiguous on variation diffs. In summary, our contributions are:

Formalization We present variation trees and variation diffs as formalizations for variability in source code and edits to it. We prove that variation diffs are sound and complete.

Classification We present a catalog of classes for edits to variability in source code and prove its completeness and unambiguity.

Automation We present `DiffDetective`, a tool to automatically classify edits in histories of software in which variability is implemented with the C preprocessor.

Validation We validate that our concepts and classes are suitable to describe and classify edits to variability in source code.

2 VARIABILITY IN SOURCE CODE

In this section, we first give an intuitive answer to what variability in source code is. Second, we propose a formalization for variability in source code, which we use to classify edits to variability in this work. Third, we discuss the suitability of our formalization.

2.1 What Is Variability in Source Code?

Variability in source code means that certain code should only be present in certain *variants* of the software. Yet, specifying variability of source code by listing the respective set of variants for each source code fragment is usually infeasible for tools and developers because configurable software may induce millions of variants [68, 92]. Instead, distinguishing variants in terms of *features* proved to be successful in software product-line engineering [4, 16, 39, 81] and clone management [28, 37, 63, 90], a feature being informally defined as a variable characteristic of the software. Each variant of a software system is then specified by a feature selection, usually referred to as *configuration* [4], stating which features the variant implements. Each source code fragment in turn, is associated to the features it implements. As common in the literature [4, 15, 18, 58, 63, 67, 90, 99], we refer to this association as the *presence condition* of the code. Source code is thus included in exactly those variants whose configurations satisfy the code's presence condition.

In practice, conditional compilation is a widely adopted strategy to induce presence conditions. Prominent examples are the Linux kernel, Busybox, and Vim that use the C preprocessor. Source code can be mapped to any propositional formula over features with the C preprocessor directives `#if`, `#ifdef`, and `#ifndef`, where `#ifdef` and `#ifndef` check whether a certain macro name is defined or not defined, respectively. Hence, `#ifdef` and `#ifndef` can be used to check whether a feature is selected or deselected, while `#if` can test complex conditions. Listing 1 shows an excerpt of graphical user interface code from Vim with preprocessor annotations. If Vim is compiled with the feature `FEAT_GUI` selected, lines 4203 and 4204 are included in the compiled variant. If the feature `FEAT_GUI` is not selected and Vim is compiled for the Windows operating system (i.e., `MSWIN` is selected), Line 4207 is included instead. In this code snippet, lines 4203 and 4204 thus share the presence condition `FEAT_GUI` but Line 4207 has the presence condition ¬`FEAT_GUI` ∧ `MSWIN`. Yet, Listing 1 is just an excerpt that is again surrounded by further preprocessor annotations, not shown here. Consequently, the actual presence conditions are even more complex.

```
4202 #ifdef FEAT_GUI
4203     if (gui.in_use)
4204         gui_mch_set_foreground();
4205 #else
4206 # ifdef MSWIN
4207     win32_set_foreground();
4208 # endif
4209 #endif
```

Listing 1: Variability specified with conditional compilation in `src/evalfunc.c` in Vim at commit ab4cece.

As illustrated in this example, there is a difference between the presence condition of a code chunk and its direct annotation, which we refer to as the code's *feature mapping*. In particular, the presence condition may be composed of one or more feature mappings, arising from nesting. As shown in Listing 1, Line 4207 is only mapped to `MSWIN` while its presence condition is also determined by the selection of `FEAT_GUI`. A presence condition thus is a conjunction of feature mappings.

In essence, feature mappings and presence conditions are a means to describe variability in source code because they allow to define any subset of variants as target for each implementation artifact. By inspecting feature mappings and edits to feature mappings, we can thus observe edits to variability in source code.

2.2 Formalization as Variation Trees

To inspect possible edits to feature mappings, we define feature mappings and presence conditions formally. We first define what can be mapped to features and how elements can be mapped to features and feature interactions. We then introduce variation trees as a formalization to represent nesting hierarchies.

Apart from source code, variability might also affect other implementation artifacts (e.g., documentation, build files, model elements) [5, 41, 60]. Even the granularity of implementation artifacts may vary in different contexts. For instance, one might interpret source code as lines of text or as an abstract syntax tree [39]: Version control systems, such as Git, usually regard source code as lines of text while developers and compilers are aware of the code's structure. For our tooling and empirical validation, we focus on

lines of source code and preprocessor-based variability. Nevertheless, we design our concepts to cover a multitude of implementation artifacts and implementation techniques for feature mappings:

Definition 2.1 (Implementation Artifact). An implementation artifact is an identifiable unit of any granularity within a software project (e.g., tokens, lines of code, model elements, or entire files).

Mapping an artifact to the features it implements can be done in several ways. For example, each artifact can be mapped to exactly one feature [94], a set of features [39], or to exactly one propositional formula over the set of features [10, 67, 90]. In this work, we map artifacts to propositional formulas, also covering mappings to single features (as formulas consisting of a single variable) or sets of features (as conjunctions of variables). An artifact mapped to a formula f is then included in exactly those variants under whose configuration f evaluates to *true*.

As illustrated in the previous Section 2.1, an artifact's presence condition depends on the artifact's location within a nesting hierarchy of feature mappings, which exhibits a tree structure. We thus introduce variation trees as a formal model for nesting hierarchies:

Definition 2.2 (Variation Tree). 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`.

Nodes in a variation tree either represent implementation artifacts or feature mappings on all its children. The root r is synthetic to group all artifacts and mappings, and may represent an entire file for example. It's formula *true* denotes that it is part of all variants, just as an annotation `#if true` does.

As an example, Figure 1 shows the variation tree induced by the annotations in Listing 1. Each node is labeled with its line number from Listing 1 as well as its code or formula, respectively. Annotations are drawn with a blue border, while nodes referencing lines of code are drawn with a black border. Consecutive lines of code within the same annotation, are grouped as a single node and labeled with the first line's number; in particular, lines 4,203 and 4,204 are grouped. Note that `#endif` directives are part of their corresponding `mapping` nodes by determining which nodes are children and which nodes are siblings of the respective `mapping`.

To model preprocessor-based variability more accurately and because of our running example, we treat `else` statements as first-class citizens in variation trees. Not all kinds of annotations do have an `else` construct though [36, 37], in which case `else` nodes can simply be omitted. In our appendix,[1] we show that we can in fact parameterize a variation tree in the set of its node types such that variation trees can support further language constructs for specifying variability, such as syntactic sugar like `elif`. In fact, we consider

[1] https://github.com/VariantSync/DiffDetective/raw/esecfse22/appendix.pdf

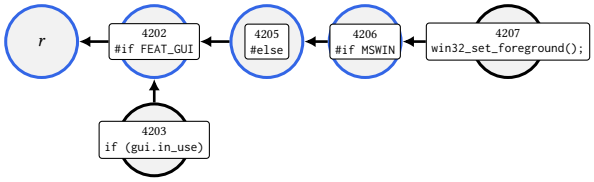


Figure 1: Variation tree of Listing 1.

`elif` directives explicitly in our validation in Section 5. Yet, such extensions are cosmetic and do not impact the validity of our results as `else` and `elif` nodes can also be expressed as `mapping` nodes (e.g., lines 4,205 and 4,206 in Listing 1 actually describe an `elif`). The set of node types must at least contain `artifact` and `mapping` and might be extended by further types to increase granularity.

Variation trees facilitate nesting nodes below `artifact` nodes. While such nesting never occurs for annotating lines of code (as lines cannot be nested in lines), other artifacts, such as abstract syntax trees [39] or models, may exhibit nesting. For example, when annotating a C struct and it's fields in an abstract syntax tree, the fields' `mapping` nodes would be placed below the `artifact` node of the struct. Yet, in our examples and our validation `artifact` nodes are line-based, so nodes will never occur below `artifact` nodes.

Variation trees directly reflect the nesting hierarchy employed by developers and thus also reflect any individual feature mapping (e.g., `#if` annotations) that is assigned to a group of artifacts. From a variation tree, we can compute the feature mapping of an artifact given its corresponding node n:

$$\mathrm{F}(n) := \begin{cases} \mathrm{F}(p(n)), & \tau(n) = \texttt{artifact}, \\ l(n), & \tau(n) = \texttt{mapping}, \\ \neg\,\mathrm{F}(p(n)), & \tau(n) = \texttt{else}\,. \end{cases} \tag{1}$$

First, the feature mapping of an `artifact` node is given by the annotation above it. Second, a `mapping` node represents a feature mapping and has its formula stored in its label that is obtained by l. Finally, for `else` branches, we have to negate the feature mapping of the corresponding `mapping`.

Based on variation trees, we can also define presence conditions. As for feature mappings, the presence condition of an implementation artifact can be computed from its corresponding node n in the variation tree. Computing the presence condition of a node is similar to computing its feature mapping, except that we have to inspect every ancestor above the node, instead of just the nearest `mapping` node. Given a variation tree with root r, we define the presence condition of a node n recursively as:

$$\mathrm{PC}(n) := \begin{cases} \mathrm{PC}(p(n)), & \tau(n) = \texttt{artifact}, \\ \mathrm{F}(n) \wedge \mathrm{PC}(p(n)), & \tau(n) = \texttt{mapping}, n \neq r, \\ \mathrm{F}(n) \wedge \mathrm{PC}(p(p(n))), & \tau(n) = \texttt{else}, \\ \mathrm{F}(n), & n = r. \end{cases} \tag{2}$$

First, as the feature mapping of an `artifact` node n is given by its parent, also its presence condition is determined by its parent $p(n)$. Second, the presence condition of a `mapping` node is given by its own formula in conjunction with the presence condition of its parent because any feature mapping may itself be nested

within further mappings. Third, also the presence condition of an `else` node is given by its own feature mapping in conjunction with the presence condition of any outer nodes. However, to access the outer nodes of an `else`, we have to retrieve the parent $p(p(n))$ of its corresponding `mapping` node $p(n)$. Fourth, the root's presence condition is solely determined by its own formula $\mathrm{F}(r) = l(r) = \mathit{true}$ because the root is not nested. Note that *true* acts neutral on conjunctions $\wedge$, and thus the synthetic root has no effect on computing presence conditions.

2.3 Discussion of Completeness and Soundness

The key property of variation trees is to distinguish presence conditions from feature mappings. Variation trees directly reflect the nesting hierarchy employed by developers and thus also reflect any individual feature mapping that is assigned to a group of artifacts. In this section, we discuss our design decisions for defining variation trees and why we consider variation trees to be an adequate model for variability in source code.

Completeness. We first discuss design decisions regarding the completeness of variation trees. That is, their suitability for expressing feature mappings and presence conditions, independent of the underlying implementation strategy for variability.

Variation trees are directly inspired by annotative approaches, in particular preprocessor directives. Yet in general, feature mappings are specified either by *annotating* source code, as shown in our example, or by *composition* of modules [39]. In annotation-based strategies, source code is annotated with the features it implements, for example with comments [12, 37], external metadata [39], or conditional compilation as shown in Listing 1. These strategies annotate source code with features or feature formulas and because our formalization does not mention any concrete macros, we argue that `mapping` nodes are general enough to cover any kind of conditional annotation. Moreover, when entire files or directories are annotated (e.g., with build systems [20]) also the directory structure with its annotation can be described with corresponding `artifact` and `mapping` nodes above the actual file contents. Compositional strategies group artifacts in modules (e.g., plugins or packages) and in turn map modules to features. Compositional strategies usually do not exhibit nesting hierarchies, so presence conditions are usually equal to feature mappings. Nevertheless, we argue that variation trees are general enough to also reflect compositional feature mappings, in particular by representing a module m assigned to feature f by a node v with type $\tau(v) =$ `mapping` and label $l(v) = f$ and grouping the implementation artifacts of m in corresponding child nodes. In conclusion, while implementation strategies for variability differ vastly in their technical realization, all of them yield the same conceptual result: a specification of feature mappings and presence conditions to determine which code fragments should be present in which variants, the key purpose of variation trees.

With variation trees, we map artifacts to propositional formulas. In practice though, artifacts might also be mapped to formulas including non-boolean expressions (e.g., arithmetics such as `#if` $x == 3$). Indeed, our concepts also support more sophisticated theories than propositional logic (e.g., higher-order logic) because we only require the propositional operators conjunction $\wedge$ and negation $\neg$, as well as the atom *true* with their usual semantics and do not impose any other restrictions on the logic.

```
 #ifdef FEAT_GUI
      if (gui.in_use)
+     {
      gui_mch_set_foreground();
-#else
-# ifdef MSWIN
+     return;
+     }
+#endif
+#if defined(MSWIN) && (!defined(FEAT_GUI) || defined(VIMDLL))
      win32_set_foreground();
-# endif
 #endif
```

Listing 2: Edits made to Listing 1 in commit `afde13b` in Vim.

To the best of our knowledge, we thus believe that variation trees are general enough to cover all strategies known to us for specifying feature mappings and presence conditions.

Soundness. We now discuss design decisions regarding the soundness of variation trees, meaning that variation trees indeed describe feature mappings and presence conditions.

Our Definition 2.2 of variation trees ensures that feature mappings and presence conditions are always defined. Each node has exactly one parent, except for the root, because variation trees are trees. Hence, the parent p is always determined in the respective cases of F and PC. By definition, every `else` has a corresponding non-root `mapping` node as its parent, hence (1) the feature mapping F of an `else` is indeed the negation of its corresponding `mapping` node, and (2) also the grandparent $p(p(n))$ exists for each `else` node, which is accessed for computing the presence condition. Further, trees and thus variation trees contain no cycles meaning that any recursive computation of feature mapping F or presence conditions PC eventually ends at the root and thus terminates. Therefore, both feature mapping F and presence condition PC can always be computed, meaning that variation trees are always a description of variability in implementation artifacts, and thus sound.[2]

Conclusion. Having laid the formal foundations for variability in source code, we inspect how the variability may be edited in the remainder of this paper. What kinds of edits could be made to Listing 1? How would such edits affect the corresponding variation tree shown in Figure 1?

3 EDITS TO VARIABILITY IN SOURCE CODE

While source code itself may be edited, also the set of variants of one or more code fragments may be altered [49, 71]. This means edits may introduce, remove, or change the feature mapping or presence condition of code fragments. For example, it might be necessary to add new code to a feature to extend it, or to remove parts of a feature mapping's formula when the formula was wrong.

[2] Variation trees may still describe illegal variants with respect to the annotated language. For example, annotated code in Listing 1 could be syntactically incorrect, yielding invalid C programs for some configurations. While correctness may be obtained by making annotations aware of the annotated language (e.g., regarding syntax- [39, 41] or type-correctness [38, 40, 47]), the variability specified by variation trees is sound.

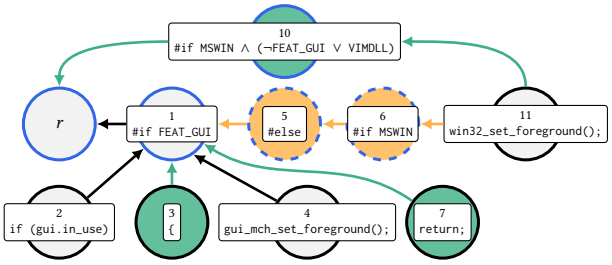


Figure 2: Variation diff of edits made in Listing 2.

Listing 2 shows changes that were made to Vim's code in Listing 1 in commit afde13b. The edit is displayed in the universal diff format, known from version control systems: Added lines are marked green and preceded by +; removed lines are marked orange and preceded by -. In this edit, source code was inserted (lines 3, 7, and 8), annotations got deleted (lines 5, 6, and 12), and replaced (lines 9 and 10). While the existing Line 11 remains untouched, its annotation was replaced. In particular, Line 11 had the presence condition ¬FEAT_GUI ∧ MSWIN before the edit but is assigned MSWIN ∧ (¬FEAT_GUI ∨ VIMDLL) afterwards. The set of variants including Line 11 thus grows as FEAT_GUI does not necessarily have to be deselected for Line 11 to be included in a variant anymore. In this section, we propose a formalism for representing such edits to variability which we use in Section 4 to classify edits to variability.

3.1 Formalization as Variation Diffs

To classify changes to variability in source code, we may inspect changes to source code and its feature mappings, as explained in Section 2. To inspect changes, we need a model that captures (1) *which* elements were edited, (2) *how* elements were edited, and (3) how the variability of each element changed. While universal diffs, as shown in Listing 2, are widely adopted for expressing *which* and *how* elements got edited, they do not reflect changes to variability well. In particular, to classify edits to variability, developers and tools have to know how feature mappings as well as entire presence conditions changed.

Based on our variation trees, we thus propose *variation diffs* as a model to describe edits to feature mappings. Analogous to universal diffs, which describe which lines of text were inserted or removed, variation diffs describe which nodes got added to or removed from a variation tree. Thus, a variation diff describes exactly two variation trees at once, one being valid before the edit and one being valid after the edit. The key idea of variation diffs is to model edits to the hierarchy of feature mappings agnostic of the actual underlying implementation artifacts.

As an example, Figure 2 shows the variation diff of the edit from Listing 2. The variation diff represents the edits made to the variation tree shown in Figure 1 corresponding to the edited source code from Listing 1. Analogous as for variation trees, we collapse continuous lines of source code with the same type of change into a single `artifact` node (e.g., lines 7 and 8). Green nodes are inserted into the variation tree, orange nodes with a dashed border are removed, and gray nodes remain unchanged. Orange edges are present before the edit, and green edges exist after the edit. For example, Line 11 (on the right in Figure 2) is nested below the annotations at lines 6 and 5 before the edit but is placed within the new annotation at Line 10 after the edit.

As variation diffs describe edits to variation trees, we formalize variation diffs as an extension (i.e., a generalization) of variation trees to graphs in which nodes and edges may also be edited:

Definition 3.1 (Variation Diffs). 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, node labels l, and a function $\Delta : V \cup E \to \{+, -, \bullet\}$ that defines if a node or edge was added $+$, removed $-$, or unchanged $\bullet$, such that project(D, t) is a variation tree for all times $t \in \{\mathrm{b}, \mathrm{a}\}$.

We refer to the variation trees before and after the edit induced by a variation diff as the *projections* of the variation diff. To reason about projections, we use the time values b (before) and a (after). We may obtain the projection of a variation diff $(V, E, r, \tau, l, \Delta)$ at time $t \in \{\mathrm{b}, \mathrm{a}\}$ by including only those nodes and edges that exist at time t:

$$\begin{aligned}\text{project}((V, E, r, \tau, l, \Delta), t) \coloneqq (&\{v \in V \mid \text{exists}(t, \Delta(v))\},\\ &\{e \in E \mid \text{exists}(t, \Delta(e))\},\\ &r, \tau, l).\end{aligned}$$

A node or edge exists before (b) the edit if it was not added, and exists after (a) the edit if it was not removed. Formally, a node or edge $x \in V \cup E$ with diff type $d = \Delta(x)$ exists at time $t \in \{\mathrm{b}, \mathrm{a}\}$ if:

$$\text{exists}(t, d) \coloneqq (t = \mathrm{b} \wedge d \neq +) \vee (t = \mathrm{a} \wedge d \neq -).$$

We can obtain the feature mapping and presence condition of a node in a variation diff before or after the edit from the respective projection. For a variation diff D, we refer to the feature mapping $\mathrm{F}(n)$ of a node n in the projection project(D, t) at time $t \in \{\mathrm{b}, \mathrm{a}\}$ as $\mathrm{F}_t(n)$. Analogously, we abbreviate the presence condition of the node n in the respective projection as $\mathrm{PC}_t(n)$.

An example for a projection can be found in Figure 1 which is the projection of the variation diff from Figure 2 before the edit (i.e., Figure 1 = project(Figure 2, b)). The variation tree contains exactly the unchanged ($\bullet$) and removed ($-$) nodes and edges from the diff. Inserted ($+$) nodes and edges are only present after the edit.

3.2 Completeness and Soundness

We prove that variation diffs are complete and sound with respect to possible edits to variation trees.

Completeness. We prove that variation diffs are *complete*, meaning that they can describe any edit to any variation tree. Every edit to a variation tree transforms it from an old state to a new state. By considering the old state to be removed entirely, and the new state to be inserted, any edit can be expressed solely using insertions and deletions. Variation diffs allow to mark any subset of nodes and edges to be added or removed via Δ. Naively, we can diff two variation trees by defining $\Delta(o) = -$ for all nodes and edges o of the old variation tree and $\Delta(n) = +$ for all nodes and edges n of the new variation tree. Therefore, variation diffs are complete with respect to variation trees. Formally, we also have to prove that when building a variation diff this way, the projections of the variation diff yield exactly the original two variation trees. This proof is part of our appendix in our replication package.[1]

Soundness. Variation diffs are *sound* by construction, meaning that any variation diff describes an actual edit to a variation tree. By definition, we can obtain both projections from a variation diff to ensure that the diff holds enough information to actually represent all differences between two variation trees. Thus, any variation diff describes an edit to a variation tree.

Conclusion. We proposed variation diffs as a complete and sound model for edits to variability of implementation artifacts. While variation trees model nesting hierarchies employed by developers or tools to structure feature mappings, variation diffs identify changes to individual feature mappings within the nesting hierarchies. To this end, variation diffs do not only cover changes to source code but also to feature mappings. In the following, we use variation diffs to classify edits.

4 CLASSIFYING EDITS TO VARIABILITY

Our main goal is to classify every edit to variability in source code. With variation diffs as a complete model, classifying edits to variability reduces to inspecting possible structures in variation diffs. The key observation is that we can classify the edit made to the variability of an artifact, by inspecting its diff type Δ and the location of its corresponding node in the variation diff. Hence, edits made to an artifact can be described locally. To classify edits, we propose nine edit classes that inspect the variation diff locally, from the perspective of an `artifact` node. We prove that our classes are complete (i.e., every node is in at least one class) and unambiguous (i.e., every node is in at most one class).

A variation diff may describe changes to many artifacts simultaneously, such as in Figure 2. A composition of instances of edit classes in turn, may emerge into a more meaningful change, that we refer to as composite edit. We show that our class definitions are indeed basic building blocks to describe more complex changes.

Key to our classification is its customizability. While we propose a catalog of classes based on use cases from variation control-systems [90], managed clone-and-own [10], and variability-aware mutation testing [2], other classifications might suit other use cases. We show that our classification scheme easily allows to define alternative classifications that remain complete and unambiguous.

4.1 Catalog of Edit Classes

Our definition of edit classes is inspired by the edit patterns by Stănciulescu et al. [90] that are designed to reflect relevant edits for variation control systems [59]. Their patterns are regular expressions over C preprocessor annotated code, which we found to be incomplete (i.e., some edits cannot be classified) and ambiguous (i.e., some edits match more than one pattern).

We describe each class as a propositional predicate on `artifact` nodes within a variation diff. The formalization as propositional predicates has the benefits that (1) it is easily reproducible (e.g., no sophisticated graph matching is required), (2) satisfiability solvers are very effective reasoning engines and part of many analysis tools already, and (3) completeness and unambiguity can be ensured by construction. Note that the classes only classify `artifact` nodes as these represent actual implementation artifacts of which we want to observe how their variability has changed.

We distinguish edit classes for added, removed, and unchanged artifacts. While an artifact itself might be unchanged, its variability might have changed, described by its ancestors in the corresponding variation diff. For further use, we define three predicates $\text{added}(v) \coloneqq (\Delta(v) = +)$, $\text{removed}(v) \coloneqq (\Delta(v) = -)$, and $\text{unchanged}(v) \coloneqq (\Delta(v) = \bullet)$ that each evaluate to *true* if a given node v has the corresponding diff type.

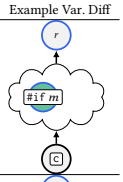
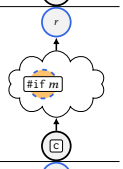
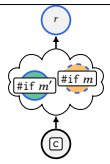
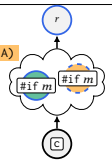
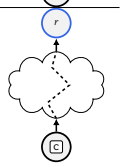
We present the formal definition of each edit class, as well as an example edit to preprocessor-based source code and a sketch of its corresponding variation diff in Table 1. The shown text-based diff and variation diff serve as examples and are not part of a class' definition. In the following, we discuss each class.

AddWithMapping and AddToPC. Our first two classes cover the insertion of source code. Depending on whether the code was inserted with or without a new feature mapping, we identify the edit as an instance of the *AddWithMapping* or *AddToPC* class respectively. Distinguishing insertion with or without a new feature mapping is relevant for variation control systems in which a newly introduced feature mapping is used to integrate the edited source code into an internal representation of the code base [28, 90]. *AddWithMapping* classifies an `artifact` node c that was inserted together with a new `mapping` node $M_a(c)$, where the function $M_t(c)$ returns the node that defines the feature mapping of an `artifact` node c at time t (i.e., $M_t(c) \coloneqq M_t(p_t(c))$ iff $\tau(c) = \texttt{artifact}$ and otherwise $M_t(c) \coloneqq c$). We have to use M_t because `artifact` nodes may be nested below `artifact` nodes and thus just inspecting parent nodes is insufficient. In conditional compilation, *AddWithMapping* corresponds to inserting code with preprocessor annotations. For *AddToPC*, inserted code is not associated with a new feature mapping but is placed below an already existing mapping $M_a(c)$.

RemWithMapping and RemFromPC. Our classes for deletions of source code are dual to our classes for addition: For removed code, we also distinguish whether the code's feature mapping was removed or not. Dual to *AddWithMapping*, the class *RemWithMapping* contains removed `artifact` nodes c whose feature mapping $M_b(c)$ was also removed. Dual to *AddToPC*, the class *RemFromPC* contains plain deletions of source code without modification to a node's feature mapping.

Generalization and Specialization. For unchanged code, we investigate if and how its variability changed. In particular, unchanged code has a presence condition PC_b before the edit and a presence condition PC_a after the edit. Depending on how these presence conditions relate, the set of variants of the source code might have changed in different ways. For example, Node 11 in Figure 2 is unchanged but its presence condition changed from $\neg\texttt{FEAT\_GUI} \wedge \texttt{MSWIN}$ to $\texttt{MSWIN} \wedge (\neg\texttt{FEAT\_GUI} \vee \texttt{VIMDLL})$. Thus, the set of variants of Node 11 grew because `FEAT_GUI` does not necessarily have to be deselected anymore. We refer to such an edit, as a *Generalization*. Formally, the set of variants of the node c grows iff its set of variants before the edit is a subset of the variants after the edit and not vice versa. The set of variants of c before the edit is a subset of the variants after the edit if and only if $\text{PC}_b(c) \Rightarrow \text{PC}_a(c)$ is a tautology, which we write as $\text{PC}_b(c) \models \text{PC}_a(c)$. We can check whether this formula is a tautology with a SAT solver via $\neg\,\text{SAT}(\neg(\text{PC}_b(c) \Rightarrow \text{PC}_a(c)))$. The intuition behind requiring a tautology is that c should remain in *all* variants it was previously included in. When instead, the set of variants of an unchanged

Table 1: Edit classification.

Edit Class Definition	Example Diff	Example Var. Diff	Edit Class Definition	Example Diff	Example Var. Diff	Edit Class Definition	Example Diff	Example Var. Diff
AddWithMapping(c) := added(c) ∧ added($M_b(c)$)	+ #if m + c + #endif		*AddToPC*(c) := added(c) ∧ ¬ added($M_a(c)$)	#if m + c #endif		*Specialization*(c) := unchanged(c) ∧¬($PC_b(c) \models PC_a(c)$) ∧ ($PC_a(c) \models PC_b(c)$)	+ #if m c + #endif	
RemWithMapping(c) := removed(c) ∧ removed($M_b(c)$)	- #if m - c - #endif		*RemFromPC*(c) := removed(c) ∧ ¬ removed($M_b(c)$)	#if m - c #endif		*Generalization*(c) := unchanged(c) ∧ ($PC_b(c) \models PC_a(c)$) ∧¬($PC_a(c) \models PC_b(c)$)	- #if m c - #endif	
Reconfiguration(c) := unchanged(c) ∧¬($PC_b(c) \models PC_a(c)$) ∧¬($PC_a(c) \models PC_b(c)$)	- #if m + #if m' c #endif		*Refactoring*(c) := unchanged(c) ∧($PC_b(c) \models PC_a(c)$) ∧($PC_a(c) \models PC_b(c)$) ∧($path_b(c) \neq path_a(c)$)	- #if A || (B && !A) + #if A || B c #endif		*Untouched*(c) := unchanged(c) ∧($PC_b(c) \models PC_a(c)$) ∧($PC_a(c) \models PC_b(c)$) ∧($path_b(c) = path_a(c)$)		

`artifact` node c shrinks, we refer to the change as a *Specialization*. A *Specialization* occurs, when the presence condition becomes more restrictive. A simple example in conditional compilation is the insertion of a non-superfluous[3] directive whose condition m has to be satisfied for the code c to be included. In the variation diff, this example corresponds to the insertion of a new `mapping` node somewhere above the `artifact` node c.

Reconfiguration. In case the set of variants is in no subset relation, we speak of a *Reconfiguration*, as the set of variants changed in an arbitrary way. This happens for example, when a non-superfluous condition is replaced by a different non-superfluous condition, or when different directives are added or removed simultaneously, such that the set of variants after the edit is neither a subset nor a superset of the previous set of variants.

Refactoring. Besides, the set of variants might not have changed at all. In this case, the presence conditions might have been refactored (e.g., an annotation $A \vee (B \wedge \neg A)$ was simplified to $A \vee B$) or not changed at all. In case the presence condition was refactored, it remained semantically equivalent but its ancestors in the variation diff have changed. If the ancestors changed, the paths $path_b(c)$ and $path_a(c)$ from a node c to the root before and after the edit are different, where $path_t(c)$ describes the path from the node c to the root of the variation diff at time $t \in \{b, a\}$ (i.e., $path_t(c) = c$ iff c is the root, and $path_t(c) = (c, path_t(p_t(c)))$ otherwise).

Untouched. The variability of a piece of source code might neither be changed semantically nor syntactically. This means, the presence condition remained equivalent regarding $\Rightarrow$ and the paths before and after the edit from a node to the root did not change (e.g., Line 1 in Listing 2 is *Untouched*). While no edit occurred for an `artifact` in this case, we include *Untouched* to be complete.

4.2 Completeness and Unambiguity

Our edit classes are complete (i.e., every `artifact` node is in at least one class) and unambiguous (i.e., every `artifact` node is in at most one class). Our classes are unambiguous as they are mutually exclusive, and they are complete as at least one predicate evaluates to *true* for each possible evaluation (i.e., the disjunction of the predicates is a tautology). We prove the completeness and unambiguity of our classes in our appendix.[1]

Given that our classes are complete on variation diffs and that variation diffs are complete with respect to possible edits to variation trees (cf. Section 3.2), we conclude that our classes are a complete classification of edits to variation trees. In Section 2.3, we discussed our design decision for variation trees to be as complete as possible in representing variability, in particular of source code. Thus, assuming that variation trees are complete, our classes are a complete classification of edits to variability in source code. Further, whenever a class occurs in a variation diff, that occurrence indeed classifies an edit to variability because variation diffs and variation trees are sound.

4.3 Defining Other Classifications

A key benefit of classifying edits via predicates over `artifact` nodes in variation diffs is that we can build other classifications, while proving completeness and unambiguity requires only minor adaptions to existing proofs. We constructed our catalog of classes based on use cases in some variational systems [2, 10, 90], but other classifications may be required to serve other use cases.

In particular, classifying edits based on predicates, gives rise to infinitely many possible classifications, each based on another set of predicates. We can always split a class, defined by a predicate $\psi(c)$, into two new classes $\psi_1(c) := \psi(c) \wedge \varphi(c)$ and $\psi_2(c) := \psi(c) \wedge \neg\varphi(c)$ by adding a new clause φ that distinguishes a further case. Such a predicate φ always exists, because there are infinitely many variation diffs and each predicate may inspect another property of the diff. As an example, we could split the *Refactoring* class to also inspect whether the two presence conditions are syntactically equal to see whether only the presence condition's distribution across feature mappings was changed or if the formula was altered.

In our appendix, we first describe a proof scheme for proving the completeness or unambiguity of any given classification that is

[3] A superfluous condition does not influence the variability [4, 93], such as `#if true`.

based on a set of predicates. We then apply these schemes to our catalog of edit classes to prove it to be complete and unambiguous. Thus, for a new classification, the schemes can be reused to prove the respective properties, while the proofs for our catalog serve as examples on how to apply the schemes.

4.4 Discussion

While our classes are unambiguous for `artifact` nodes in variation diffs, diffs in general are ambiguous as they can be constructed in multiple ways. For example, it might have been more intuitive to show the removed lines 5 and 6 after the insertion of lines 9 and 10 to group related changes in Listing 2. We identified two sources of ambiguity: propositional formulas and the specification of diffs.

Ambiguity in Feature Mappings. A conjunction $A \wedge B$ of two feature mappings can be stored in a single `mapping` node with label $A \wedge B$ or in two nested nodes labeled with A and B, respectively. This ambiguity is subject to the developer's choice. For example, in conditional compilation developers are free to annotate a piece of code with `#if defined(A) && defined(B)` or with two nested annotations `#ifdef A` and `#ifdef B`. Yet, this ambiguity is not a limitation but a key quality of variation diffs. In fact, we designed variation diffs to reflect hierarchies of annotations to retain the ontological information given by developers to better understand how variability is described in practice.

Ambiguity in Diffing. While variation diffs are complete and sound regarding variation trees, they remain ambiguous. For example, as discussed in Section 3.2, one could describe an edit to a variation tree as a variation diff in which all nodes and edges of the old tree are removed and all new nodes and edges are inserted. Such a description of an edit is rarely useful though, and so nodes and edges being present before and after the edit should be considered unchanged. However, detecting whether a node is unchanged is not unambiguously determined (e.g., when lines of code are moved). In fact, ambiguity in expressing edits to implementation artifacts is a limitation of any differencing technique in general, sometimes requiring advanced matching heuristics [25, 44, 74]. We thus share this limitation with existing research on differencing and edit classification. Yet, we can control this ambiguity by employing a deterministic diffing technique, that yields the same variation diff for the same two input revisions of a source code file. We have implemented a deterministic parser for constructing variation diffs from a *unix diff* for our validation in Section 5.

4.5 Composite Edits

Although our classification is complete, variability in source code might witness more complex changes. For instance, Stănciulescu et al. [90] observed that when adding code with a feature mapping, developers sometimes simultaneously add code to variants not including the feature with an `#else` branch (called *AddIfdefElse*). While such an edit is classified by our catalog as a simultaneous application of two *AddWithMapping* edits, this classification loses their connection in the graph structure. In particular, earlier studies observe or investigate different kinds of complex *patterns* to increase the accuracy when evaluating research [2, 10, 90]. However, existing studies face overlaps (i.e., an edit might be matched by more than one pattern) leading to ambiguity.

Variation diffs enable us to systematically derive complex patterns by interpreting them as a composition of class definitions. We thus refer to such complex patterns as *composite* edit patterns. While composite patterns might overlap, the overlap is explicitly accounted for by their composition of classes that do not overlap.

As our classes are complete, unambiguous, and fine-grained, we argue that they indeed serve as a set of building blocks for complex patterns. In our appendix, we show that all edit patterns reported in previous studies [2, 90] are either a composite edit pattern or equivalent to one of our classes.[1] A catalog of composite edit patterns can be useful but is out of scope of this paper.

5 VALIDATION

In this section, we validate that our theoretical results can be transferred to practice. Therefore, we classify the edits made in the development histories of 44 real-world software product lines. Our theory is based on the assumptions that (1) variation trees are complete and sound (cf. Section 2.3), and (2) feature mappings are syntactically correct. To determine whether these assumptions are valid in practice, we validate the completeness of variation diffs, by verifying that every patch with syntactically correct mappings can be parsed to a variation diff, and by inspecting how often mappings are syntactically correct (RG1). We validate the proofs for completeness and unambiguity of our classification, by verifying that each edited line of source code is in exactly one edit class (RG2). Moreover, we inspect the relevance of each class by observing its occurrence in practice (RG3). Finally, we validate that our classification can be automated and scales such that variation diffs could be parsed and our classes could be detected by future variability management tools (RG4). In summary, our research goals are:

RG1 Validate the completeness of variation diffs as a representation for edits to variability in source code.
RG2 Validate that our edit classes are complete and unambiguous.
RG3 Validate that our edit classes are relevant (i.e., all classes occur in practice).
RG4 Validate that edit classification can be automated and scales.

5.1 Subject Systems

For our validation, we analyze 44 open-source software product lines from different domains. We include all 40 software product lines whose variability was analyzed by Liebig et al. [56] regarding their complexity, granularity and types of extensions applied by preprocessor directives. Fortunately, all systems are still publicly available, and we provide updated links in our replication package. While these 40 systems already cover a wide spectrum of domains – including but not limited to webservers, operating systems, database systems, antivirus, media players, and editors – we decided to also include four other systems to further increase external validity: Busybox, a collection of tools for embedded systems that is widely studied in research on configurable software [20, 31, 35, 43, 46, 58, 79, 83]; Marlin, a 3D printer firmware from which edit patterns were retrieved semi-automatically by Stănciulescu et al. [90] (cf. Section 6); libssh, an SSH library also studied in the context of configurable systems [67]; and the game engine Godot as a new domain. In Table 2, we present an overview of the four subject systems with the longest revision history as

Table 2: Overview and results of the four largest and four new subject systems of the 44 analyzed systems.

Name	Domain	#total commits	#processed commits	#diffs	#artifact nodes	AddToPC	AddWithMapping	RemFromPC	RemWithMapping	Specialization	Generalization	Reconfiguration	Refactoring	runtime	avg. runtime per processed commit	median runtime per processed commit
linux	operating system	1,072,601	870,429	1,875,864	12,483,635	51.5%	0.9%	45.8%	0.9%	0.2%	0.3%	0.2%	0.1%	10,829.9s	12.2ms	6ms
freebsd	operating system	272,207	179,753	729,831	9,747,920	49.4%	2.7%	43.2%	2.3%	0.7%	0.8%	0.6%	0.3%	5,418.2s	29.6ms	5ms
gcc	compiler framework	191,405	122,777	416,750	3,040,492	50.1%	1.6%	45.9%	1.3%	0.4%	0.4%	0.3%	0.1%	6,023.5s	47.4ms	21ms
emacs	text editor	154,155	37,943	71,321	571,109	47.6%	3.3%	43.5%	2.7%	1.0%	1.0%	0.5%	0.5%	1,349.1s	33.0ms	14ms
⋮						36 other systems										
godot	game engine	40,944	18,867	107,845	1,267,555	48.4%	3.3%	42.7%	3.0%	0.5%	0.4%	1.4%	0.3%	2,687.2s	141.8ms	7ms
marlin	3d printing	19,260	14,607	84,332	567,164	38.0%	12.2%	35.0%	8.7%	0.6%	1.6%	2.9%	1.0%	549.0s	37.4ms	10ms
busybox	embedded systems	17,447	14,485	41,146	393,324	45.8%	3.7%	43.4%	3.6%	1.0%	0.7%	1.2%	0.5%	244.8s	16.8ms	6ms
libssh	network	5,352	4,439	8,465	56,440	52.0%	2.5%	41.9%	1.7%	1.1%	0.4%	0.2%	0.2%	21.3s	4.8ms	3ms
total	–	2,594,912	1,708,111	4,900,820	45,413,708	49.8%	2.1%	44.7%	1.7%	0.5%	0.5%	0.4%	0.2%	37,558.1s	21.5ms	7ms

well as the four new subject systems (we omit the remaining 36 systems for brevity). The full results for all systems are part of our appendix.[1] Notably, we analyze the entire revision history of the Linux kernel which to the best of our knowledge has the largest history in the domain of configurable systems with more than one million commits. All subject systems are managed with the version control system git.

5.2 Experiment Setup

To pursue our research goals, we built `DiffDetective`,[4] a library and command-line tool to retrieve variation diffs from git histories of preprocessor-based product lines. Each commit in a git repository consists of a set of patches, where each patch comprises all edits made to exactly one file, as shown in Listing 2. `DiffDetective` extracts all patches that are valid for our validation. A patch is valid if it modified a source code file (`.c`, `.cpp`, `.h`, or `.hpp`) with at least one non-whitespace change, and stems from a non-merge commit because merge commits do not have a single parent. Each valid patch is then parsed to a variation diff. We therefore convert local patches to full patches (i.e., we use the entire file as context and not just the changed fraction as shown in Listing 2), to account for annotations that surround edits. To pursue RG1, `DiffDetective` reports the cause whenever a patch cannot be parsed. In the "#processed commits" column of Table 2, we show how many commits contain at least one valid patch that could be parsed, about 1.7 million in total. In each parsed variation diff, we then remove all non-edited subtrees as these subtrees do not represent edits. `DiffDetective` then determines the edit class of each `artifact` node of each variation diff to validate that every node is in exactly one class (RG2) and counts their occurrences (RG3). For RG4, we measure the time it takes to process each commit and repository.

`DiffDetective` uses the FeatureIDE library [48] to reason on propositional formulas paired with the Sat4j SAT solver [55] to classify edits. We implemented the Tseytin transformation [52, 97] for larger SAT queries. To interact with git, we use the JGit library. `DiffDetective`'s parser for patches to variation diffs is explained in a bachelor's thesis [98].

We perform our validation on an Ubuntu 20.04.3 LTS system with 64-bit architecture and an Intel® Xeon® CPU E5-260v3 with 2.40Ghz clock rate. The system has 32 threads that we use to process parts of each history in parallel but a single commit is always processed in a single thread. To reduce the impact of other processes, the machine was not in use for other tasks in parallel.

[4]GitHub: https://github.com/VariantSync/DiffDetective/tree/esecfse22
DOI: 10.5281/zenodo.7110095

5.3 Results and Discussion

We summarize our results in Table 2. In total, we processed about 1.7 million commits containing about 4.9 million variation diffs in about 10.4 h with a speed of 21.5 ms per commit on average and 7 ms as median. Because of multithreading, we had an effective runtime of about 70 min yielding an effective average runtime of about 2.4 ms per commit. We classified about 45 million edits to source code (i.e., artifact nodes in variation diffs). For each edit class, we list its relative share of the `artifact` nodes.

RG1: Variation Diff Validity. Whenever `DiffDetective` cannot parse a patch to a variation diff, it reports an error and its reason. We found the only reported errors to be syntax errors in preprocessor annotations. In particular, we found 4,393 patches in which an `#endif` was missing, 1,336 patches with conditional directives without an expression, 2,732 patches with an `#endif` without an `#if`, 483 patches with an `#else` or `#elif` without a corresponding `#if`, 62 patches with an `#else` following another `#else`, and 7 patches with syntax errors in the definition of multi-line macros. In total, 0.18% of all patches were syntactically invalid. Upon a manual investigation of some of these failures, we found that the preprocessor directives were indeed invalid and were often fixed in the following commit. Thus, we find that we can parse all syntax-correct patches to variation diffs.

RG2: Validity of Edit Classes. `DiffDetective` determines the edit class for each line of code in the parsed variation diffs. In case, an `artifact` node is an instance of no or more than one class, `DiffDetective` crashes by design and reports that node. Yet, no crash occurred for any of the 44 input repositories, thus validating our proofs for completeness and unambiguity in Section 4.2 and the correctness of our implementation.

RG3: Relevance of Edit Classes. We report the occurrence counts for each class in Table 2. We omit *Untouched* as it never occurred. This is (1) expected because we removed non-edited subtrees from all variation diffs, and (2) desired because we aim to inspect edits to variability and *Untouched* describes an artifact that experiences no changes. *AddToPC* and *RemFromPC* are by far the most frequent edit classes with 49.8% and 44.7% of edits, respectively. Here, it is important that we collapse subsequent lines of

source code with the same diff type to a single `artifact` node (cf. Section 3.1) to consider the line edits as a single edit, rather than one edit per line. In line with previous research [37, 49, 90], we thus find that edits to source code are much more frequent than edits to annotations. The remaining edit classes make up for only 5.5% of the edits, which are still 2.5 million edits. Insertion and deletion of code with a feature mapping (*AddWithMapping* and *RemWithMapping*) are more frequent than edits to the feature mappings of existing source code. Edits to the variability of existing code are about equally distributed with *Refactoring* being the least common. While a *Refactoring* is the rarest edit it still occurs ~91K times. We conclude that all edit classes are relevant in practice.

RG4: Automation and Scalability. `DiffDetective` runs fully automatically and took about 10.4 h to process all commits but only 70 min in total because of multithreading. Classifying all edits in a commit requires 21.5 ms on average and 7 ms as median. We find that 99.89% of commits were processed in less than a second. All remaining commits, apart from three outliers, were processed in less than one minute. The three outliers stem from Godot and required 5, 6 and 27 min, respectively. The longest to process commit was 8c1731b. We found a single of its 65 patches to be responsible: The generated file `tools/editor/doc_data_compressed.h` with about 100K lines of code experienced 127,704 changes according to Github. The same file was edited in the other two commits. Such files could be excluded from analyses and we suppose a long process time for huge changes to be reasonable. We thus conclude that classifying edits can be automated and scales well.

5.4 Threats to Validity

Our tool `DiffDetective` could have bugs that impact our results. We tested all crucial functionality with unit tests and structured our development along issues and merge requests. To validate the correctness of our variation diff parser, we performed manual testing and implemented a consistency check that is performed on each parsed diff. This check reported no inconsistencies.

Our approach to measure relevance by counting class occurrences could be insufficient. Yet, we find that each system with at least 108 processed commits contains all of our classes. Only in five systems some classes are absent, three of which had small histories (< total 125 commits) without any valid patches.

We parse conditions to boolean logic although also non-boolean formulas occur (cf. Section 2.3) which would require more sophisticated reasoning engines than SAT solvers (e.g., SMT solvers), which may impact our performance results. Upon parsing, we replace non-boolean sub-expressions by unique variables, a technique known as *boolean abstraction*, employed in SMT solving [9], and state-of-the-art when analyzing variability at a large scale [32, 42, 58].

Our parser does not expand macro invocations and include directives which might impair the soundness of our results. Yet, sophisticated variability-aware parsing may take several minutes and up to multiple hours for a single version [32, 42, 66] and requires to tree-difference two parsed versions, which also is not yet variability-aware (cf. Section 6). As in existing research [58], we thus abstract macro invocations as constant values (e.g., we treat `FOO(3, 4)` as a constant `FOO_3_4`). We found parsing Unix diffs to variation diffs to be the best trade-off between scalability and accuracy.

Our subject systems may not be representative for the evolution of variability in source code. In particular, all subjects are open-source C/C++ repositories. To address this threat, we chose systems that were previously studied [56] and extended them by four new systems to cover a wide spectrum of about 30 different domains. Moreover, our validation focused on preprocessor-based variability by design which is inherently mostly used in C/C++ software.

6 RELATED WORK

Our work is inspired by the work of Stănciulescu et al. [90], who empirically and semi-automatically extracted a set of edit patterns from the history of two software product lines, to evaluate their model of a variation control system. When inspecting the patterns, we found them to be (1) ambiguous as they lack a formal description or tooling, (2) incomplete because some patterns miss their inverse operation, and (3) overlapping (i.e., an edit may belong to multiple patterns). To address ambiguity and incompleteness, we refined and extended their patterns with formal definitions in terms of classes. To address the overlap, we distinguish between classes and composite edit patterns. Moreover, we provide open-source tooling to automatically classify edits and our validation covers 44 systems instead of two.

Complete and unambiguous classifications of edits to variability are only available for the *problem space* so far (e.g., for feature models defining the set of valid configurations) but not the solution space (i.e., source code and feature mappings). Thüm et al. [95] introduce four groups of edits to feature models. Interestingly, our classes *Specialization*, *Generalization*, *Reconfiguration*, and *Refactoring* acting on source code correspond to these groups. This is not surprising because feature models and feature mappings both describe a set of valid configurations but for *the entire product line* or *a certain implementation artifact*, respectively. Inspired by this analogy, we decided to partly adopt the naming scheme of Thüm et al. for the above mentioned classes. Bürdek et al. [13] extend the work by Thüm et al. with concrete edit operations. Analogous to our work, Bürdek et al. distinguish between elementary edits and complex, composite edits. Thus, the classification by Bürdek et al. can be seen as the counterpart for feature models to our work on source code. Patterns for refactorings and generalizations [3] or specializations [17] to feature models do not classify edits. The complete and unambiguous classifications for edits to feature models serve as inspiration for our work but cannot be used for edits to source code and feature mappings.

Automatic analyses of edits to source code do not facilitate a complete classification. With an automatic analysis whether edits in Linux affect the feature model, feature mappings, or source code, Kröher et al. [49] inspect *that* but not *how* variability information was edited. Dintzner et al. [21] present FEVER which parses the history of product lines automatically. While FEVER retrieves extensive data and metadata for co-evolution of variability, it does not provide a classification of edits. Based on the refinement theory [11, 27], safe-evolution templates [69, 84] describe possible scenarios how feature model, feature mappings, and source code can be changed without altering the set of valid products. While the refinement theory can model any edit to variability, the described templates are incomplete by design because they target (partial)

refactorings. Moreover, there is no distinction between feature mappings and presence conditions. Nieke et al. [71] extend the notion of refinements to guide the adaption of configurations upon complex edit scenarios, such as feature merges. Nieke et al. identify three exemplary (i.e., incomplete) evolution templates. Further analyses of the evolution of product lines [96] do not classify edits. Opposed to the state of the art, we performed a large-scale validation by classifying 45 million edits from 1.7 million commits in 44 systems. We could classify all edits and proved that our classes are complete.

Semi-automatic analyses of edits to source code do not facilitate a complete classification. With a manual analysis of 657 commits, Passos et al. [78] extracted edit patterns from the evolution of Linux, but remain incomplete as they report edits that did not match any pattern. Seidl et al. [89] propose evolution operators for model-based product lines that have to be employed manually during development and make no arguments on completeness. Due to the fact that our analysis is fully automated, we could classify far more commits and systems from different domains.

Edit specifications for other purposes than a classification of edits are incomplete or ambiguous. To evaluate the benefits of recording feature mappings, Ji et al. [37] provide an extensive catalog including macroscopic edits such as cloning whole code bases. Yet, their patterns are given in natural language and thus matching patterns cannot directly be automated. While designed for testing, the mutation operators by Al-Hajjaji et al. [2] exhibit some similarity with our classes. Yet, the operators remain incomplete (e.g., *AddWithMapping* and thereby a true subset of edits is missing). Further work on refactorings [26, 57] presents proactive evolution operators similar to Seidl et al. [89] but without classifying edits. Potentially, our classes can also be used for the above purposes.

The choice calculus [23] is a formal language for variation and closely related to our variation trees. While choices model alternative variation (i.e., exactly one of a set of alternatives must be chosen), variation trees model optional variation (i.e., subtrees can be in- or excluded). While choice calculus was applied in practice [8, 90, 100, 101], it was never applied to evolution and we found optional variation to model preprocessor-based variability more naturally. In the future, we aim to study the relationship between choice calculus and variation trees.

Differencing techniques describe changes to data, such as source code in concrete [7, 14, 74] or abstract syntax [19, 22, 24, 29, 30, 33], but are unaware of variability. Our theory and our tool `DiffDetective` make diffs variability-aware by parsing a generic diff (e.g., Unix diffs) into a variation diff, which explicitly models edited variability. Thus, any diff tool may be used to obtain a generic, intermediate diff in `DiffDetective`. Semantic differencing [6, 54, 64, 72, 73, 75, 76] relies on operational semantics (e.g., control flow [6]) of single-variant systems and is oblivious to variability. Variability-aware parsing [32, 42] for conditional compilation yields a single abstract syntax tree with variability information but does not consider edits. Medeiros et al. [65] semi-automatically apply variability-aware parsing to the versions of eight systems across ~51K commits but do not inspect edits to variability. To inspect edits, tree diffing could be employed but is not yet variability-aware. Moreover, while Medeiros et al. [65] report no timing results, it is known [32, 42] that parsing a single version of Linux takes more than ten hours, while we extract all relevant information from a single commit in 21.5 ms on average.

Use Cases. Our variation trees and diffs serve as a unified, formal language to express, communicate, and reason about variability and edits to it. Research and tools on evolution of variation based on our classification are guaranteed to be complete regarding edits. For instance, our catalog increases the applicability of variability-aware mutation testing [2] by completing their existing set of mutation operators. For variation-control systems [59, 60, 90] and managed clone-and-own [45, 63], which inspect and operate primarily on edits, our classes can serve as a design reference and evaluation dataset [10, 90]. Methods for synchronizing variants [45, 63, 80, 85, 86] in managed clone-and-own [82] are guaranteed to be able to deal with any input edit from users, when based on our complete variation diffs and classes. Moreover, software developers can observe the effect of their edits on variability to, for example, validate that an edit was a refactoring or specialization, such that no further test cases have to be introduced. As a library, `DiffDetective` enables researchers and practitioners to study the evolution of variability in source code by retrieving variation diffs and detecting edit classes automatically.

7 CONCLUSION

We propose a classification of edits to variability in source code that we prove to be complete (i.e., every edit is classified by at least one class) and unambiguous (i.e., every edit is classified by at most one class). We introduce variation trees, a formal model to describe variability in source code, and variation diffs, a formal model to describe changes to variation trees, which we prove to be complete and which is sound by construction.

To validate the suitability and potential for automation of our concepts and classification, we classified about 45 million edits to source code fully automatically. In effectively 70 min, we processed about 1.7 million commits from the histories of 44 open-source software systems. 99.89% of the considered commits were processed in less than a second, making our method feasible in practical scenarios, such as continuous integration. We found that 0.2% of the patches submitted by developers contain syntactically incorrect preprocessor annotations. All other edits were classified and each of our edit classes occurs in practice.

Our edit classes can be used to gain insights into the evolution of configurable software systems and the effects of edits on variability. In the future, we plan to derive composite edit patterns from frequent compositions of instances of our edit classes which serve as building blocks to describe more complex changes.

ACKNOWLEDGMENTS

We thank our reviewers for their constructive feedback. We thank Benjamin Moosherr, Kevin Jedelhauser, and Tobias Heß for their help with the experimental setup and artifact. We thank Chico Sundermann and Jeffrey M. Young for advice on SAT solving and our proofs. We thank Alexander Boll and Sebastian Krieter for proof-reading, and Sven Apel for helpful discussions. This work has been partially supported by the German Research Foundation within the project *VariantSync* (TH 2387/1-1 and KE 2267/1-1).

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. *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 *VaMoS*. ACM, 81–88.

[3] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos José Pereira de Lucena. 2006. Refactoring Product Lines. In *GPCE*. ACM, 201–210.

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

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

[6] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2007. JDiff: A Differencing Technique and Tool for Object-Oriented Programs. 14, 1 (2007), 3–36.

[7] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Massimiliano Di Penta. 2013. LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines. In *ICSM*. IEEE, 230–239.

[8] Parisa Ataei, Fariba Khan, and Eric Walkingshaw. 2021. A Variational Database Management System. In *GPCE*. ACM, 29–42.

[9] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Technical Report. Department of Computer Science, The University of Iowa.

[10] Paul Maximilian Bittner, Alexander Schultheiß, Thomas Thüm, Timo Kehrer, Jeffrey M. Young, and Lukas Linsbauer. 2021. Feature Trace Recording. In *ESEC/FSE*. ACM, 1007–1020.

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

[12] Quentin Boucher, Andreas Classen, Patrick Heymans, Arnaud Bourdoux, and Laurent Demonceau. 2010. Tag and Prune: A Pragmatic Approach to Software Product Line Implementation. In *ASE*. ACM, 333–336.

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

[14] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. 2009. Ldiff: An Enhanced Line Differencing Tool. In *ICSE*. IEEE, 595–598.

[15] Krzysztof Czarnecki and Michal Antkiewicz. 2005. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *GPCE*. Springer, 422–437.

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

[17] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Formalizing Cardinality-Based Feature Models and Their Specialization. *SPIP* 10 (2005), 7–29.

[18] Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *GPCE*. ACM, 211–220.

[19] Michael John Decker, Michael L. Collard, L. Gwenn Volkert, and Jonathan I. Maletic. 2020. srcDiff: A Syntactic Differencing Approach to Improve the Understandability of Deltas. *JSEP* 32, 4 (2020).

[20] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. A Robust Approach for Variability Extraction from the Linux Build System. In *SPLC*. ACM, 21–30.

[21] 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. *EMSE* 23, 2 (2018), 905–952.

[22] Georg Dotzler and Michael Philippsen. 2016. Move-Optimized Source Code Tree Differencing. In *ASE*. ACM, 660–671.

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

[24] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *ASE*. 313–324.

[25] Yuanrui Fan, Xin Xia, David Lo, Ahmed E. Hassan, Yuan Wang, and Shanping Li. 2021. A Differential Testing Approach for Evaluating Abstract Syntax Tree Mapping Algorithms. In *ICSE*. IEEE, 1174–1185.

[26] Wolfram Fenske, Thomas Thüm, and Gunter Saake. 2014. A Taxonomy of Software Product Line Reengineering. In *VaMoS*. ACM, 4:1–4:8.

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

[28] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *ICSE*. IEEE, 665–668.

[29] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *TSE* 33, 11 (2007), 725–743.

[30] Veit Frick, Thomas Grassauer, Fabian Beck, and Martin Pinzger. 2018. Generating Accurate and Compact Edit Scripts Using Tree Differencing. In *ICSME*. IEEE, 264–274.

[31] Paul Gazzillo. 2017. Kmax: Finding All Configurations of Kbuild Makefiles Statically. In *ESEC/FSE*. ACM, 279–290.

[32] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *PLDI*. ACM, 323–334.

[33] Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *WCRE*. 279–288.

[34] Wolfgang Heider, Rick Rabiser, Paul Grünbacher, and Daniela Lettner. 2012. Using Regression Testing to Analyze the Impact of Changes to Variability Models on Products. In *SPLC*. ACM, 196–205.

[35] Tobias Heß, Chico Sundermann, and Thomas Thüm. 2021. On the Scalability of Building Binary Decision Diagrams for Current Feature Models. In *SPLC*. ACM, 131–135.

[36] Patrick Heymans, Quentin Boucher, Andreas Classen, Arnaud Bourdoux, and Laurent Demonceau. 2012. A Code Tagging Approach to Software Product Line Development. *STTT* 14 (2012), 553–566. Issue 5.

[37] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability with Embedded Annotations. In *SPLC*. ACM, 61–70.

[38] Christian Kästner and Sven Apel. 2008. Type-Checking Software Product Lines—A Formal Approach. In *ASE*. IEEE, 258–267.

[39] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *ICSE*. ACM, 311–320.

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

[41] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don Batory. 2009. Guaranteeing Syntactic Correctness for All Product Line Variants: A Language-Independent Approach. In *TOOLS Europe*, Manuel Oriol and Bertrand Meyer (Eds.). Springer, 175–194.

[42] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *OOPSLA*. ACM, 805–824.

[43] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A Variability-Aware Module System. In *OOPSLA*. ACM, 773–792.

[44] Timo Kehrer, Udo Kelter, Pit Pietsch, and Maik Schmidt. 2012. Adaptability of Model Comparison Tools. In *ASE*. ACM, 306–309.

[45] Timo Kehrer, Thomas Thüm, Alexander Schultheiß, and Paul Maximilian Bittner. 2021. Bridging the Gap Between Clone-and-Own and Software Product Lines. In *ICSE*. IEEE, 21–25.

[46] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is There a Mismatch Between Real-World Feature Models and Product-Line Research?. In *ESEC/FSE*. ACM, 291–302.

[47] Sergiy Kolesnikov, Alexander von Rhein, Claus Hunsen, and Sven Apel. 2013. A Comparison of Product-Based, Feature-Based, and Family-Based Type Checking. In *GPCE*. ACM, 115–124.

[48] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. 2017. FeatureIDE: Empowering Third-Party Developers. In *SPLC*. ACM, 42–45.

[49] Christian Kröher, Lea Gerling, and Klaus Schmid. 2018. Identifying the Intensity of Variability Changes in Software Product Line Evolution. In *SPLC*. ACM, 54–64.

[50] Jacob Krüger and Thorsten Berger. 2020. Activities and Costs of Re-Engineering Cloned Variants Into an Integrated Platform. In *VaMoS*. ACM, Article 21, 10 pages.

[51] Jacob Krüger and Thorsten Berger. 2020. An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse. In *ESEC/FSE*. ACM, 432–444.

[52] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2022. Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *ASE*. ACM. To appear.

[53] Elias Kuiter, Jacob Krüger, Sebastian Krieter, Thomas Leich, and Gunter Saake. 2018. Getting Rid of Clone-and-Own: Moving to a Software Product Line for Temperature Monitoring. In *SPLC*. ACM, 179––189.

[54] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *CAV*. Springer, 712–717.

[55] Daniel Le Berre and Anne Parrain. 2010. The Sat4j Library, Release 2.2. *JSAT* 7, 2-3 (2010), 59–64.

[56] 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 *ICSE*. IEEE, 105–114.

[57] Jörg Liebig, Andreas Janker, Florian Garbe, Sven Apel, and Christian Lengauer. 2015. Morpheus: Variability-Aware Refactoring in the Wild. In *ICSE*. IEEE,

380–391.
[58] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *ESEC/FSE*. ACM, 81–91.
[59] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *GPCE*. ACM, 49–62.
[60] Lukas Linsbauer, Alexander Egyed, and Roberto Erick Lopez-Herrejon. 2016. A Variability Aware Configuration Management and Revision Control Platform. In *ICSE*. ACM, 803–806.
[61] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability Extraction and Modeling for Product Variants. *SoSyM* 16, 4 (2017), 1179–1199.
[62] Sascha Lity, Manuel Nieke, Thomas Thüm, and Ina Schaefer. 2019. Retest Test Selection for Product-Line Regression Testing of Variants and Versions of Variants. *JSS* 147 (2019), 46–63.
[63] Wardah Mahmood, Daniel Strueber, Thorsten Berger, Ralf Laemmel, and Mukelabai Mukelabai. 2021. Seamless Variability Management With the Virtual Platform. In *ICSE*. IEEE, 1658–1670.
[64] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2010. A Manifesto for Semantic Model Differencing. *MODELS*, 194–203.
[65] Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. 2013. Investigating Preprocessor-Based Syntax Errors. In *GPCE*. ACM, 75–84.
[66] Gabriela K. Michelon, Wesley K. G. Assunção, David Obermann, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. 2021. The Life Cycle of Features in Highly-Configurable Software Systems Evolving in Space and Time. In *GPCE*. ACM, 2–15.
[67] Gabriela Karoline Michelon, David Obermann, Lukas Linsbauer, Wesley Klewerton Guez Assunção, Paul Grünbacher, and Alexander Egyed. 2020. Locating Feature Revisions in Software Systems Evolving in Space and Time. In *SPLC*. ACM, Article 14, 11 pages.
[68] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. In *SPLC*. ACM, 289–301.
[69] Laís Neves, Paulo Borba, Vander Alves, Lucinéia Turnes, Leopoldo Teixeira, Demóstenes Sena, and Uirá Kulesza. 2015. Safe Evolution Templates for Software Product Lines. *JSS* 106 (2015), 42–58.
[70] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulesza, and Paulo Borba. 2011. Investigating the Safe Evolution of Software Product Lines. In *GPCE*. ACM, 33–42.
[71] Michael Nieke, Gabriela Sampaio, Thomas Thüm, Christoph Seidl, Leopoldo Teixeira, and Ina Schaefer. 2022. Guiding the Evolution of Product-Line Configurations. *SoSyM* 21 (2022), 225–247. Issue 1.
[72] Yannic Noller, Hoang Lam Nguyen, Minxing Tang, Timo Kehrer, and Lars Grunske. 2021. Complete Shadow Symbolic Execution with Java PathFinder. *SEN* 44, 4 (2021), 15–16.
[73] Yannic Noller, Corina S. Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. 2020. HyDiff: Hybrid Differential Software Analysis. In *ICSE*. ACM, 1273–1285.
[74] Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto. 2020. How Different are Different Diff Algorithms in Git? *EMSE* 25, 1 (2020), 790–823.
[75] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a Doubt: Testing for Divergences Between Software Versions. In *ICSE*. ACM, 1181–1192.
[76] Nimrod Partush and Eran Yahav. 2014. Abstract Semantic Differencing via Speculative Correlation. In *OOPSLA*. ACM, 811–828.
[77] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. 2013. Feature-Oriented Software Evolution. In *VaMoS*. ACM, 1–8.
[78] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of Variability Models and Related Software Artifacts. *EMSE* 21, 4 (2016).
[79] Tobias Pett, Sebastian Krieter, Tobias Runge, Thomas Thüm, Malte Lochau, and Ina Schaefer. 2021. Stability of Product-Line Sampling in Continuous Integration. In *VaMoS*. ACM, Article 18, 9 pages.
[80] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. 2016. Synchronizing Software Variants with VariantSync. In *SPLC*. ACM, 329–332.
[81] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.
[82] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2013. Managing Cloned Variants: A Framework and Experience. In *SPLC*. ACM, 101–110.
[83] Sebastian Ruland, Lars Luthmann, Johannes Bürdek, Sascha Lity, Thomas Thüm, Malte Lochau, and Márcio Ribeiro. 2018. Measuring Effectiveness of Sample-Based Product-Line Testing. In *GPCE*. ACM, 119–133.
[84] Gabriela Sampaio, Paulo Borba, and Leopoldo Teixeira. 2019. Partially Safe Evolution of Software Product Lines. *JSS* 155 (2019), 17–42.
[85] Thomas Schmorleiz and Ralf Lämmel. 2016. Similarity Management of 'Cloned and Owned' Variants. In *SAC*. ACM, 1466–1471.
[86] 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 *ICSME*. IEEE. To appear.
[87] Sandro Schulze, Oliver Richers, and Ina Schaefer. 2013. Refactoring Delta-Oriented Software Product Lines. In *AOSD*. ACM, 73–84.
[88] Sandro Schulze, Thomas Thüm, Martin Kuhlemann, and Gunter Saake. 2012. Variant-Preserving Refactoring in Feature-Oriented Software Product Lines. In *VaMoS*. ACM, 73–81.
[89] Christoph Seidl, Florian Heidenreich, and Uwe Aßmann. 2012. Co-Evolution of Models and Feature Mapping in Software Product Lines. In *SPLC*. ACM, 76–85.
[90] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. 2016. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *ICSME*. IEEE, 323–333.
[91] Chico Sundermann, Michael Nieke, Paul Maximilian Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2021. Applications of #SAT Solvers on Feature Models. In *VaMoS*. ACM, Article 12, 10 pages.
[92] Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2020. Evaluating #SAT Solvers on Industrial Feature Models. In *VaMoS*. ACM, Article 3, 9 pages.
[93] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *EuroSys*. ACM, 47–60.
[94] Sahil Thaker, Don Batory, David Kitchin, and William Cook. 2007. Safe Composition of Product Lines. In *GPCE*. ACM, 95–104.
[95] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning About Edits to Feature Models. In *ICSE*. IEEE, 254–264.
[96] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Towards Efficient Analysis of Variation in Time and Space. In *VariVolution*. ACM, 57–64.
[97] Grigori S. Tseytin. 1983. *On the Complexity of Derivation in Propositional Calculus*. Springer, 466–483.
[98] Sören Viegener. 2021. *Empirical Evaluation of Feature Trace Recording on the Edit History of Marlin*. Bachelor's Thesis. University of Ulm.
[99] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-Condition Simplification in Highly Configurable Systems. In *ICSE*. IEEE, 178–188.
[100] Eric Walkingshaw and Klaus Ostermann. 2014. Projectional Editing of Variational Software. In *GPCE*. ACM, 29–38.
[101] Jeffrey M. Young, Paul Maximilian Bittner, Eric Walkingshaw, and Thomas Thüm. 2022. Variational Satisfiability Solving: Efficiently Solving Lots of Related SAT Problems. *EMSE* (2022). To appear.
[102] Shurui Zhou, Ştefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. 2018. Identifying Features in Forks. In *ICSE*. ACM, 105–116.