Views on Edits to Variational Software

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

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

Sandra Greiner
sandra.greiner@unibe.ch
University of Bern
Bern, Switzerland

Benjamin Moosherr
benjamin.moosherr@uni-ulm.de
University of Ulm
Ulm, Germany

Sebastian Krieter
sebastian.krieter@uni-ulm.de
University of Ulm
Ulm, Germany

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

Software systems are subject to frequent changes, for example to fix bugs or meet new customer requirements. In variational software systems, developers are confronted with the complexity of evolution *and* configurability on a daily basis; essentially handling changes to many distinct software variants simultaneously. To reduce the complexity of configurability for developers, filtered or projectional editing was introduced: By providing a partial or complete configuration, developers can interact with a simpler *view* of the variational system that shows only artifacts belonging to that configuration. Yet, such views are available for individual revisions only but not for *edits* performed across revisions. To reduce the complexity of evolution in variational software for developers, we extend the concept of views to edits. We formulate a correctness criterion for views on edits and introduce two correct operators for view generation, one operator suitable for formal reasoning, and a runtime optimized operator. In an empirical study, we demonstrate the feasibility of our operators by applying them to the change histories of 44 open-source software systems.

CCS CONCEPTS

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

KEYWORDS

software variability, software evolution, software product lines, projectional editing, variation control

ACM Reference Format:
Paul Maximilian Bittner, Alexander Schultheiß, Sandra Greiner, Benjamin Moosherr, Sebastian Krieter, Christof Tinnes, Timo Kehrer, and Thomas Thüm. 2023. Views on Edits to Variational Software. In *27th ACM International Systems and Software Product Line Conference - Volume A (SPLC '23), August 28-September 1, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3579027.3608985

1 INTRODUCTION

Developing variational software confronts developers with complexity in two dimensions [3, 80]. First, developers face *variation in time* as the software varies with each revision over the course of its development (e.g., to fix bugs). Second, *variation in space* denotes that the software is configurable (e.g., via conditional compilation), requiring developers to deal with multiple software variants simultaneously. Combined, both dimensions require developers to reason on edits that affect potentially millions of software variants at once [56, 78].

To manage both dimensions of variability, variation control systems [50] offer workflows [4] based on projectional editing[1] [31, 34, 77, 85], also known as filtered editing [16, 50, 68] or view-based editing [5]. The central idea is that developers can edit a complex variational system in terms of a *view* that shows only a subset of the system relevant for the current task. To checkout a view for editing, developers provide a complete [23, 68] or partial configuration [77, 85] of the configuration options existing in the variational system at a certain revision. Developers can modify the view and commit it against a description of the edited features — usually a partial configuration but a single feature in the simplest case [4, 50]. Consequent updates of the underlying complex software are performed automatically and hidden from the user.

While projectional editing reduces the complexity of modifying variational software, it does not allow for performing views on edits in the revision history. For instance, code reviewers may want to focus on the features they are responsible for. When backporting new bugfixes to older but stable versions of the software, it is necessary to adopt only the subset of changes that can be applied, as, for example, in the development of the Linux kernel [72]. However, when multiple features are edited in the same revision, developers are confronted with all changes at once.

[1]Considered in the mathematical sense where a projection builds a view, not in the model-driven engineering sense where the abstract syntax is edited.

To the best of our knowledge, reducing the complexity of edits to variability in terms of projectional editing has not been addressed yet. Semantic history slicing [44, 70, 71, 88] analyzes the history of a program, for example, to find semantically related commits, or to identify all commits that contributed to a bug or certain lines of code. Commit untangling [15, 45, 58, 61, 73, 87] and similar techniques identify which development concerns are addressed by a set of changes with the goal of grouping the changes by the concern they address. Untangling is performed based on a broad spectrum of heuristic criteria such as dependencies between changes, text similarity, location in a file, or overlap of changes between commits but not for possible variability. Feature revisions [5, 49] allow for selecting a feature in a specific revision but not for performing a view on the edit which may have introduced or modified it.

In this paper, we contribute the theoretical foundations for views on edits to variational software and demonstrate empirically that such views can be computed fast in practice. We observe that views on edits can be described in terms of views on single revisions. Based on this observation, we formalize a correctness criterion for views on edits. We propose two algorithms for view generation and prove their correctness: One algorithm that emerges from the correctness criterion and is suitable for formal reasoning, and a runtime-optimized algorithm. In an empirical study on 44 real-world and open source software product lines, we study the feasibility of view generation practice. In summary, we contribute:

View Types that generalize state-of-the-art views on revisions to other relevant types of views apart from partial configuration (Section 4),

Theoretical Foundations in terms of a correctness criterion and two functions for generating views on edits to variational software (Section 5), and a

Feasibility Study that demonstrates that generating views on edits to variational software can be generated automatically and fast (Section 6).

2 PRELIMINARIES

Based on an exemplary user story, we motivate practical use cases for views on edits to variational software. Afterwards, we explain how we describe variability and edits to it as a basis for formalizing views on edits in the following sections.

2.1 Motivating Example

Consider a team of C++ developers that implements variational software through conditional compilation [6]. The software includes a method `prepend` for a linked list, shown in Listing 1. The method `prepend` inserts a given element `e` as the new list head. C preprocessor annotations, starting with `#`, conditionally in- or exclude source code from compilation. Developers obtain a variant of the software by specifying values for each preprocessor annotation, being `Ring` and `DoubleLink` here. For instance, introducing the statement `#define DoubleLink 0` excludes Lines 9–14 from the resulting C++ program whereas selecting the feature through the statement `#define DoubleLink 1` includes them. Hence, the feature `DoubleLink` ensures elements being doubly-linked upon insertion, and selecting feature `Ring` optionally turns a list into a ring such that the last element is linked to the first one.

```
1  void prepend(T e) {
2    Itm* newHead = new Itm(e);
3    newHead->suc = head;
4  #if Ring
5    if (empty())
6      last = newHead;
7  #endif
8    last->suc = newHead;
9  #if DoubleLink
10   head->prev = head;
11   #if Ring
12     newHead->prev = last;
13   #endif
14 #endif
15   head = newHead;
16 }
```

Listing 1: Variability implemented with the C preprocessor in a C++ method.

```
1    void prepend(T e) {
2      Itm* newHead = new Itm(e);
3      newHead->suc = head;
4    #if Ring
5      if (empty())
6        last = newHead;
7  -#endif
8      last->suc = newHead;
9  +#endif
10   #if DoubleLink
11 +   if (head) {
12 -   head->prev = head;
13 +     head->prev = newHead;
14 +   }
15     #if Ring
16       newHead->prev = last;
17     #endif
18   #endif
19     head = newHead;
20   }
```

Listing 2: Three bug fixes to the code in Listing 1.

```
 void prepend(T e) {
   Itm* newHead = new Itm(e);
   newHead->suc = head;
 #if Ring
   if (empty())
     last = newHead;
-#endif
   last->suc = newHead;
+#endif
   head = newHead;
 }
```

Listing 3: Bob's view

```
 void prepend(T e) {
   Itm* newHead = new Itm(e);
   newHead->suc = head;
-  last->suc = newHead;
 #if DoubleLink
+  if (head) {
-  head->prev = head;
+    head->prev = newHead;
+  }
 #endif
   head = newHead;
```

Listing 4: Charlotte's view

Imagine the developer Alice detects and fixes bugs in Listing 1 with the patch shown in Listing 2. First, the scope of the feature `Ring` is wrong: The successor of the last element is set to the first element, even in variants without feature `Ring`. Alice fixes this bug by moving the `#endif` in Line 7 to Line 9 thereby extending the scope of the annotation `#if Ring` to include `last->suc = newHead`. Second, for empty lists, the method crashes in feature `DoubleLink` as `head->prev` is undefined. Thus, Alice inserts a check whether the head exists in Lines 11 and 14. Third, she fixes the wrong double link by removing Line 12 and inserting Line 13. These changes affect *multiple features and derivable variants* of the software.

Suppose a second developer Bob is responsible for the feature `Ring` but is not experienced in `DoubleLink`. As Alice's edits affected multiple features, it may not be obvious to Bob how his feature `Ring` changed. In a code review, Bob wants to focus on and see changes only to `Ring`, particularly as commits may encompass more patches than just a single one. Listing 3 presents the desired view on Alice's edit that only includes the feature `Ring`. Now, for example, Bob can run tests of variants in- or excluding only that feature.

Assume a third developer Charlotte is in charge of deploying older but stable revisions of the software to customers. Charlotte considers Alice's fixes crucial to those stable revisions. Yet, the feature `Ring` does not exist in the stable revisions. Applying Alice's patch directly is impossible because the patch changes lines that do not exist. Alice's patch could be applied to the stable revisions by removing the changes to Ring. Listing 4 represents this patch, which is a view on Alice's patch excluding the code of feature Ring. In this patch, Line 4 appears as deleted although Alice's edit did not change it. Alice moved the line to feature Ring, thereby *re*moving it from its prior annotation. Requiring recent changes in stable revisions is known as *patch backporting* — a crucial task, for instance, in developing the Linux kernel [72].

In summary, both, Bob and Charlotte, face issues that may be solved by *views* on a patch to variational software. While we employ a fictive running example for brevity and focus, similar and more complex cases arise in practice and have been studied in the past [14, 29, 50, 77, 85]. Developers often edit multiple features in a single commit [54, 82] and the C-preprocessor is used at a large-scale to implement variability [29, 46]. In fact, the sometimes excessive usage of preprocessor annotations is referred to as the #ifdef-hell [22, 42, 51, 53]. Concluding, views on edits address the need to assist the maintenance of large variational software.

2.2 Background on Software Variability

Introducing variability aims at reusing source code by identifying common and different parts among software variants, typically expressed in terms of *features* [6, 18, 64]. Usually, features denote distinct functionality relevant to customers [9]. Different variability mechanisms [6] map domain artifacts at different levels of granularity, such as lines of source code (as shown in Listing 1), nodes in the abstract syntax tree [33], entire files or modules (also referred to as compositional variability) [6], or model elements in model-based engineering [24], onto features or combinations thereof [10, 17, 26]. The mapping is usually referred to as the artifact's *presence condition* as it embodies a condition under which the artifact is present in a variant. In Listing 1, the presence condition of Line 5 is Ring and the presence condition of Line 12 is DoubleLink ∧ Ring. Both presence conditions may be more complex if the entire prepend method would be surrounded with further annotations. A configuration assigns truth values to each feature and serves to derive a software variant by including all artifacts whose presence condition is satisfied by the configuration, and excluding all other artifacts. For example, selecting Ring and deselecting DoubleLink includes the entire code except for Lines 10 and 12 of Listing 1. A software system implementing variability in this way is referred to as a *software product line* [6, 18]. In a software product line, a feature model [8, 30] may additionally restrict the set of valid variants, for example, to enforce dependencies or exclusion criteria between features [8].

In this paper, we focus on *static* variability as realized with conditional compilation. In a variant, no variability annotations remain and thus implementation artifacts (e.g., source code) cannot interact with variability information. Hence, variability annotations are usually unaware of the syntax or semantics of the implementation artifacts and vice versa. For instance, the C-preprocessor considers the underlying source code as plain text and is unaware of the source code's operational semantics. Conversely, when a final C/C++ program is compiled or run, no C-preprocessor annotations exist anymore because they were already resolved.

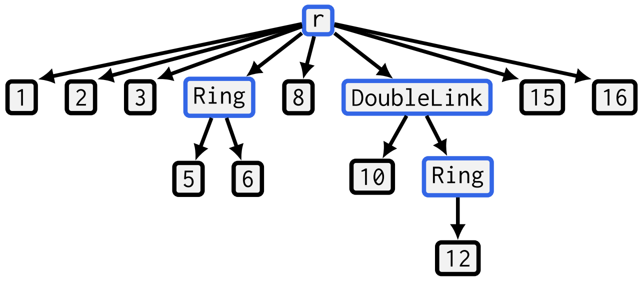

Figure 1: Variation Tree of Listing 1.

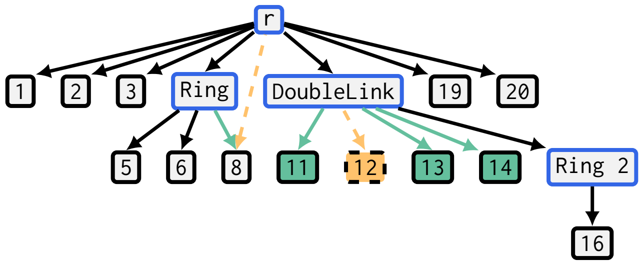

Figure 2: Variation Diff of Listing 2.

To express the variability in a file, we use variation trees [14] – a generic tree in which nodes reflect variability annotations or domain artifacts. As an example, Figure 1 represents Listing 1 as a variation tree. A *mapping* node (blue border) represents a variability annotation whereas an *artifact* node (black border) represents domain artifacts (i.e., lines of code in this example). A synthetic *root* node r groups annotated code at the file level. Labels of artifact nodes state the respective line number. As for preprocessor annotations, a mapping node maps all nodes in its subtree to a formula. Thus, variation trees reflect the nesting structure established by annotations. Note that #endifs are not part of the tree as they declare the end of an annotation's scope only. While this example shows a variation tree for a C preprocessor program, variation trees do not require a specific macro language. Hence, variation are generic and can represent different annotation languages [14].

Formally, a variation tree is a labeled, typed graph (V, E, r, τ, l) with nodes V, edges $E \subset V \times V$, root $r \in V$, typing $\tau : V \rightarrow \{\texttt{artifact}, \texttt{mapping}, \texttt{else}\}$, and label function l [14]. The label l maps each node $v \in V$, onto a domain artifact iff $\tau(v) = \texttt{artifact}$, a propositional formula over the set of features iff $\tau(v) = \texttt{mapping}$, or nothing iff $\tau(v) = \texttt{else}$. An else node may occur below a mapping node only, and a node cannot have more than one else child node. The root is defined to be neutral: $\tau(r) = \texttt{mapping}$ and $l(r) = \textit{true}$. Let $p(T, v)$ denote the parent node of a node $r \neq v \in V$ in the tree T. Let $\mathcal{T}$ denote the set of all variation trees.

The *presence condition* $\text{PC}(T, v)$ of a node $v \in V$ in a variation tree $T = (V, E, r, \tau, l)$ can be computed as follows [14]:[2]

[2]This formula is slightly adapted [14] in that it (1) inlines the auxiliary definition of feature mappings and (2) takes the variation tree T as additional argument. This is required for obtaining the parent $p(T, v)$ of a node v because only T contains the necessary information about edges.

$$\mathrm{PC}(T,v) \coloneqq \begin{cases} \mathrm{PC}(T,w), & \tau(v) = \texttt{artifact}, \\ l(v) \wedge \mathrm{PC}(T,w), & \tau(v) = \texttt{mapping}, v \neq r, \\ \neg l(w) \wedge \mathrm{PC}(T,p(T,w)), & \tau(v) = \texttt{else}, \\ \mathit{true}, & v = r \end{cases}$$
$$\textbf{where } w \coloneqq p(T,v)$$

The presence condition of an artifact node is given by the presence condition of its enclosing annotation. A non-root mapping node combines its own formula $l(v)$ with any outer presence condition $\mathrm{PC}(T,w)$. The presence condition of an else node is given by the negation of its corresponding mapping node (which is located above the else in the variation tree) as well as any outer annotations.

2.3 Background on Edits to Software Variability

To model views on edits to variability, we require a model that can describe edits to variability. Moreover, it should be general enough to capture any variability in the sense described above, and not just a single concrete mechanism such as the C preprocessor. To this end, we use *variation diffs*, a sound and complete model for edits to variability [14]. Variation diffs are sound which means that any variation diff describes an edit to a variational system, and complete which means that any change to a variational system can be expressed as a variation diff. Variation diffs express edits as a difference between two variation trees [14].

Figure 2 shows the variation diff corresponding to Alice's patch of Listing 2. Node types and labels are depicted similarly as before for variation trees but additionally, nodes and edges are colored: Gray nodes and black edges are unchanged, green nodes and edges are inserted, and orange, dotted nodes and edges are deleted.

Formally, a variation diff is a graph $D = (V, E, r, \tau, l, \Delta)$ with nodes V, edges E, root r, typing τ, and labeling l as defined for variation trees [14]. Additionally, the coloring $\Delta : V \cup E \longrightarrow \mathcal{P}(\{\mathrm{b}, \mathrm{a}\})$ denotes the set of times (or revisions) at which a node or edge exists, where $\mathcal{P}$ denotes the power set. The values b and a reference the times *before* or *after* the edit, respectively.[3]

In Figure 2, green nodes and edges are inserted, thus, they exist only after the edit (i.e., $\Delta(x) = \{\mathrm{a}\}$). Orange nodes and edges are deleted and exist only before the edit (i.e., $\Delta(x) = \{\mathrm{b}\}$). Gray nodes and black edges are unchanged and exist at all times (i.e., $\Delta(x) = \{\mathrm{b}, \mathrm{a}\}$).

The semantics of a variation diff $D = (V, E, r, \tau, l, \Delta)$ is given by the corresponding variation trees T_{b} *before* and T_{a} *after* the edit, where $T_{\mathrm{b}} = \mathit{project}(D, \mathrm{b})$ and $T_{\mathrm{a}} = \mathit{project}(D, \mathrm{a})$ with *project* [14] being defined as:

$$\begin{aligned} \mathit{project}(D,t) \coloneqq (&\{ v \in V \mid t \in \Delta(v)\}, \\ &\{ e \in E \mid t \in \Delta(e)\}, \\ &r, \tau, l). \end{aligned} \tag{1}$$

As an example, Figure 1 is the projection of Figure 2 before the edit (i.e., Figure 1 = *project*(Figure 2, b)).

[3]We adapted the definition of the coloring Δ from its original definition [14]. Originally, the coloring denoted the type of change (i.e., if a node or edge was inserted +, deleted −, or unchanged ●). Yet in fact, change types are mere names for times of existence given by the correspondence + = {a}, deleted − = {b}, or unchanged ● = {b, a}. Our adapted definition leads to simpler algorithms and proofs.

3 VIEWS ON VARIATIONAL SYSTEMS

The goal of this paper is to define views on edits to variational software. The key observation is that views on edits can be described in terms of views on the state (i.e., a single revision) of a variational software system, as we will elaborate in Section 5. In this section, we cover views on variational systems first.

The idea of a view is to act as a filter on *relevant* parts of a system. For instance, a piece of source code may be deemed relevant if it implements a certain feature. Developers decide which parts of the code they consider relevant. We model this decision on *relevance* as a binary predicate

$$\rho : \mathcal{T} \times V \longrightarrow \{\top, \bot\} \tag{2}$$

over the nodes V in a given variation tree $T = (V, E, r, \tau, l) \in \mathcal{T}$ where $\top$ and $\bot$ denote the Boolean values *true* and *false*, respectively. For a given node $v \in V$ of the given tree T, the relevance $\rho(T, v)$ decides whether v is kept in a view on T. Besides nodes, relevance predicates may have to inspect a node's location or neighborhood because of which also the tree T serves as input. We discuss useful implementations of relevance predicates ρ later in Section 4.

To obtain a view of the variation tree T, we ought to keep only those nodes that are deemed relevant by a relevance ρ. Yet, removing an arbitrary subset of nodes may destroy the variation tree, potentially leaving it in a disconnected state, which in turn would not correspond to a meaningful variational system anymore. To guarantee that all relevant nodes are part of the view and that the view is a variation tree, all relevant nodes must be connected to the rest of the tree. Hence, in a view we must include each relevant node v as well as all their ancestors. We therefore introduce the function p^+ that determines the transitive closure over the parents p, including the original node v:

$$p^+(T,v) \coloneqq \begin{cases} \{v\}, & v = r, \\ \{v\} \cup p^+(T, p(T,v)), & \text{otherwise.} \end{cases}$$

Then, the set of nodes kept in the view on a variation tree $T = (V, E, r, \tau, l)$ consists of the ancestors $p^+(T, v)$ of each relevant node $v \in V$ with $\rho(T, v)$:

$$\mathit{viewnodes}(T, \rho) \coloneqq \bigcup_{v \in V,\ \rho(T,v)} p^+(T, v). \tag{3}$$

The function *viewnodes* essentially collects all nodes in paths from a relevant node to the root (including both the relevant node $v \in V$ with $\rho(T, v)$ itself and the root in the ancestors set $p^+(T, v)$).

Finally, we can define the view on a variation tree $T = (V, E, r, \tau, l)$ as a variation tree that contains exactly those nodes *viewnodes*(T, ρ) that are relevant due to ρ or required for the tree structure:

$$\begin{aligned} &\mathit{view}_{\mathit{tree}}(T, \rho) \coloneqq (V', E', r, \tau, l) \\ \textbf{where}\quad &V' \coloneqq \mathit{viewnodes}(T, \rho), \\ &E' \coloneqq E \cap (V' \times V'). \end{aligned} \tag{4}$$

An edge can only be part of a view if both its nodes remain in the view. We thus restrict the set of edges E accordingly.

In summary, we reformulated and generalized views for the state (i.e., single revisions) of a variational system to variation trees and arbitrary relevance predicates.

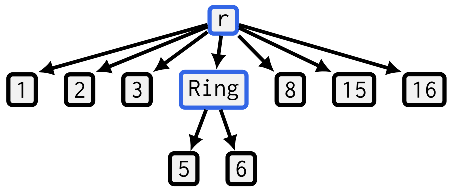

Figure 3: = $\boldsymbol{view_{tree}}$(**Figure 1**, ***configure***(**`Ring`**, ¬(**`DoubleLink`** ∧ **`Ring`**)))

4 INTERESTING VIEWS AND EXAMPLES

So far, we defined views on variational software at an *abstract* level: A user may decide which elements are relevant through the binary relevance ρ. Yet, we have not discussed whether it is realistic that users can indeed define such a relevance, and for which purposes. In this section, we present three types of views in terms of generators for relevance predicates: partial configurations, as used in the literature, feature traces, and artifact search. We demonstrate that each of these relevance operators represents typical activities when developing variational software.

4.1 Partial Configuration

The standard type of view on a variational system, is a view onto a variant or a certain subset of variants [23, 77, 85], for example all variants without a certain feature. Views of this type are the result of the ordinary configuration process of variational systems (i.e., the common semantics of variability). Given a partial configuration, which selects or deselects some or all of the features of the variational system, all artifacts whose presence condition contradicts that configuration are removed [85].

Given a partial configuration as a propositional formula c and a feature model as a propositional formula m, the relevance is given by $\rho = \mathit{configure}(c, m)$, where

$$\mathit{configure}(c, m) \coloneqq \lambda(T, v) \to \mathrm{SAT}(c \land m \land \mathrm{PC}(T, v)). \tag{5}$$

We use λ-notation to emphasize that *configure* is meant to be used as a higher-order function to serve as a generator for relevance predicates.[4] In *configure*, SAT denotes a standard satisfiability check on a propositional formula, using a satisfiability solver. The satisfiability check tests whether at least one complete configuration (i.e., assignment of all features to *true* or *false*) exists that (1) matches the partial configuration c (i.e., every selection or deselection in c is also made in the complete configuration), (2) is valid in terms of the feature model m, and (3) includes the node v with presence condition $\mathrm{PC}(T, v)$. Intuitively, *configure* tests whether at least one valid variant exists that contains node v under the partial configuration c. If a feature model is not available, m can be set to *true*, which means that there are no constraints among features. The node v is not restricted to artifacts; also annotations (i.e., `mapping` nodes) may be removed from the view.

As an example, Figure 3 is a view on the variation tree shown in Figure 1 asking for all variants that select at least feature `Ring` (i.e., the partial configuration c is a propositional formula that is only the literal `Ring`). Assume that doubly-linked rings are forbidden which

[4]Partially applying functions like this is referred to as currying [76]. To compute a view on a variation tree T, we write $\mathit{view}_{tree}(T, \mathit{configure}(c, m))$ because $\mathit{configure}(c, m)$ returns a function which can then be plugged into view_{tree}.

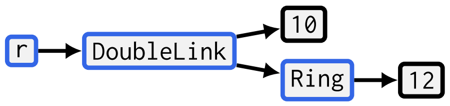

Figure 4: = $\boldsymbol{view_{tree}}$(**Figure 1**, ***trace***(**`DoubleLink`**))

means that the feature model is a formula m = ¬(`DoubleLink` ∧ `Ring`) ensuring that features `DoubleLink` and `Ring` are alternative. Thus, the relevance in this example is *configure*(`Ring`, ¬(`DoubleLink` ∧ `Ring`)). Since feature `Ring` is selected, feature `DoubleLink` will be deselected due to the feature model m. The resulting view thus contains all code except for feature `DoubleLink`.

4.2 Feature Traces

A key concern of variability mechanisms and software product lines is the traceability of features [12]. It is crucial to find locations of interest in a software system reliably and exhaustively for maintaining and comprehending a software system [74, 83]. In fact, tracing features or requirements to their implementation is essential to all stakeholders of software development and one of the most common activities of developers [11, 65, 74, 83, 86].

Using a dedicated relevance ρ, we can pinpoint exactly those locations in the software that are affected by a certain feature f (assuming locations of features are explicit as in variational software):

$$\mathit{trace}(f) \coloneqq \lambda(T, v) \to f \in \mathit{vars}(\mathrm{PC}(T, v)). \tag{6}$$

where $\mathit{vars}(\phi)$ denotes the set of all variables occurring in a formula ϕ. A relevance $\mathit{trace}(f)$ for a feature f is a syntactical search that selects a node if and only if its presence condition is affected by a certain feature f (i.e., contains that feature as a variable). A view $\mathit{view}_{tree}(T, \mathit{trace}(f))$ on a variation tree T thus shows exactly the part of the tree that is influenced by either the selection or deselection of feature f.[5]

As an example, Figure 4 is a view on the variation tree shown in Figure 1 (i.e., Figure 4 = view_{tree}(Figure 1, *trace*(`DoubleLink`))). This view traces the feature `DoubleLink`, and thus only the paths from the root to the implementation of `DoubleLink` remain.

Besides syntactical feature tracing as above, developers may be interested in semantic tracing. With the relevance $\mathit{trace}_{\supseteq}(c)$, developers can trace exactly those artifacts that require the selection or deselection of certain features, given as a partial configuration c:

$$\mathit{trace}_{\supseteq}(c) \coloneqq \lambda(T, v) \to (\mathrm{PC}(T, v) \models c),$$

where $\models$ denotes a tautology check (i.e., for all assignments under which $\mathrm{PC}(T, v)$ evaluates to *true* also c has to evaluate to *true*). A relevance $\mathit{trace}_{\supseteq}(c)$ includes exactly those nodes in the view that require the configuration c to be satisfied for them to appear in a variant. For example, a node v with presence condition $\mathrm{PC}(T, v) = A \land B$ would be included in the views described by $\mathit{trace}_{\supseteq}(A)$, $\mathit{trace}_{\supseteq}(B)$, and $\mathit{trace}_{\supseteq}(A \land B)$ but not in $\mathit{trace}_{\supseteq}(\neg A)$ or $\mathit{trace}_{\supseteq}(X)$ for another feature X. The name $\mathit{trace}_{\supseteq}$ denotes that the selections and deselections in the configuration c in fact have to be a subset of the selections and deselections required by the presence condition. When flipping the tautology to

$$\mathit{trace}_{\subseteq}(c) \coloneqq \lambda(T, v) \to (c \models \mathrm{PC}(T, v)),$$

[5]Except when f is redundant in the formula but then it is still present in the annotation and thus shown in the view.

the resulting view contains exactly the nodes that are selected by the partial configuration c (i.e., c is is complete for $\mathrm{PC}(T, v)$). Thus a node contained in a view described by $\mathit{trace}_{\subseteq}(c)$ is fully configured.

The difference between partial configuration and semantic feature traces is that partial configuration yields views on variants or subsets of variants while feature traces yield views on individual features or feature interactions. Nodes, whose inclusion or exclusion is undecided by a partial configuration, survive in a configuration process but not in feature traces. For instance, a node v with presence condition $A \wedge B$ is kept by $\mathit{configure}(C, \mathit{true})$ because selecting an unrelated feature C does not decide the appearance of v. Contrary, $\mathit{trace}_{\supseteq}(C)$ and $\mathit{trace}_{\subseteq}(C)$ hide v exactly because v is unrelated.

4.3 Artifact Search

Another interesting view for developers is the search for certain artifacts, similar to feature traces but for artifacts. For instance in C-preprocessor-based product lines, annotations might span the entire document or even multiple files. Using the following relevance generator, developers may inspect only the annotations to certain artifacts a that might be buried in a large file or system:

$$\mathit{search}(a) \coloneqq \lambda(T, v) \to (\tau(v) = \texttt{artifact} \wedge a = l(v)). \tag{7}$$

A relevance $\mathit{search}(a)$ deems only those nodes relevant that match the search artifact a. This kind of view embodies a search algorithm that finds all locations of a certain source code artifact (up to exact equality). For example, a view of the code in Listing 1 with the relevance $\mathit{search}(\texttt{last = head;})$ yields a variation tree that solely contains the root, the annotation `Ring` in Line 4, and below that annotation, Line 6 which exactly contains the search string. Weakening the required equality in the definition of *search* to look for sub-expressions or similar artifacts may enhance the search capabilities. Thus, the views with a generalized relevance are as general as to also work as a search for implementation artifacts.

In summary, we demonstrated the flexibility of abstracting over relevance predicates ρ to serve typical developer activities when developing variational software. Next, we proceed to the main goal of this paper, views on edits.

5 VIEWS ON EDITS TO VARIABILITY

We now define requirements and operators for describing and generating views on edits to variational software. The key observation is that views on edits can be described in terms of views on variational systems, described in Section 3. We start by formalizing a correctness criterion that defines what it means to be a view on an edit. We then show that our correctness criterion gives rise to a simple and elegant but potentially inefficient way to generate views on edits. Finally, we design a run-time optimized generation of views and discuss possible implementations.

5.1 Correctness Criterion

Before we can construct views on edits to variability, we have to identify the expected behavior of such a view. Recall a key purpose of views on variational systems, formalized in Section 3 and in work on projectional editing [85]: Editing a variational system can be simplified by first creating a view on the system via a relevance ρ, then editing said view, to finally apply the edit to the complex underlying system by automated lifting. While developers perform an *edit on a view* of the system, the actual (automatically computed) edit is more complex. In this sense, the developer's edit thus can be seen as a view on the actual complex edit. Moreover, a *view on the edit* with the same relevance ρ should be the edit made by the developer. Otherwise, the view would be inconsistent with the developer's edit, and may even be different each time a view is created. Thus, a *view on an edit* must be equivalent to an *edit to a view* of the system:

$$\begin{tikzcd}
T_{\mathrm{b}} \arrow[rrr, "\mathit{view}_{\mathit{tree}}(T_{\mathrm{b}}, \rho)"] \arrow[d, no head] & & & T'_{\mathrm{b}} \arrow[d, no head] \\
{} \arrow[r, "\mathit{diff}"] \arrow[d, no head] & D \arrow[lu, "\mathit{project}(D, \mathrm{b})"'] \arrow[r, "\mathit{view}_{\mathit{diff}}(D, \rho)"] \arrow[ld, "\mathit{project}(D, \mathrm{a})"] & D' \arrow[ru, "\mathit{project}(D', \mathrm{b})"] \arrow[rd, "\mathit{project}(D', \mathrm{a})"'] & {} \arrow[l, "\mathit{diff}"'] \arrow[d, no head] \\
T_{\mathrm{a}} \arrow[rrr, "\mathit{view}_{\mathit{tree}}(T_{\mathrm{a}}, \rho)"] & & & T'_{\mathrm{a}}
\end{tikzcd}$$

Figure 5: Commuting diagram for the correctness criterion for views on edits to variability as defined in Definition 5.1.

Definition 5.1 (Correctness). A view function $\mathit{view}_{\mathit{diff}}$ for edits to variability is correct if and only if

$$\begin{aligned}&\mathit{view}_{\mathit{diff}}(\mathit{diff}(T_{\mathrm{b}}, T_{\mathrm{a}}), \rho)\\ \equiv\ &\mathit{diff}(\mathit{view}_{\mathit{tree}}(T_{\mathrm{b}}, \rho), \mathit{view}_{\mathit{tree}}(T_{\mathrm{a}}, \rho))\end{aligned}$$

for any two variation trees $T_{\mathrm{b}}, T_{\mathrm{a}}$ and relevance $\rho : \mathcal{T} \times V \longrightarrow \{\top, \bot\}$, where $\equiv$ denotes semantic equivalence (cf. Definition 5.3), and *diff* is a differencing function according to the following definition.

Definition 5.2 (Difference). For any two variation trees $T_{\mathrm{b}}, T_{\mathrm{a}}$, a function $\mathit{diff} : \mathcal{T} \times \mathcal{T} \longrightarrow \mathcal{D}$ is a differencing function iff it is inverse to projections for all times $t \in \{\mathrm{b}, \mathrm{a}\}$:

$$\big(\mathit{project}(\mathit{diff}(T_{\mathrm{b}}, T_{\mathrm{a}}), \mathrm{b}) = T_{\mathrm{b}}\big) \text{ and } \big(\mathit{project}(\mathit{diff}(T_{\mathrm{b}}, T_{\mathrm{a}}), \mathrm{a}) = T_{\mathrm{a}}\big).$$

The correctness criterion (Definition 5.1) ensures that a view on that edit (left hand side) is the same as an edit to a view (right hand side). Figure 5 visualizes the correctness criterion as a commuting diagram. The goal of the diagram is to reach the view D' (right center) on the difference between two variation trees T_{b} and T_{a} (left corners) with respect to a relevance ρ. Starting from the two trees $T_{\mathrm{b}}, T_{\mathrm{a}}$, there are two ways to compute D': First, we can compute views $T'_{\mathrm{b}} = \mathit{view}_{\mathit{tree}}(T_{\mathrm{b}}, \rho)$ and $T'_{\mathrm{a}} = \mathit{view}_{\mathit{tree}}(T_{\mathrm{a}}, \rho)$ (right corners) on each tree separately and then create a variation diff $D' = \mathit{diff}(T'_{\mathrm{b}}, T'_{\mathrm{a}})$ from these tree views. Second, we can use a dedicated viewing function $\mathit{view}_{\mathit{diff}}$ that creates the view $D' = \mathit{view}_{\mathit{diff}}(D, \rho)$ from the diff $D = \mathit{diff}(T_{\mathrm{b}}, T_{\mathrm{a}})$ (center). The correctness criterion states that the viewing function $\mathit{view}_{\mathit{diff}}$ is correct if and only if both paths commute (i.e., always yield semantically equivalent result).

For the correctness criterion, it is important to *not* enforce strict equality $=$ because expressing edits is usually ambiguous [14] and subject to research on advanced matching heuristics [21, 35, 60]. For example, in Listing 2, Alice deleted an `#endif` in Line 7 and inserted it in Line 9. Alternatively, the diff could also show the `#endif` as being unchanged and Line 8 (`last->suc = newHead;`) as being removed and reinserted within the scope of feature `Ring`. Even more drastically, considering the whole old version of the `prepend` method as deleted and its new version as being inserted would still describe an equivalent edit. Thus, syntactically different diffs can represent equivalent edits in the sense that both edits produce the same result for the same input. Hence, we require *semantic equivalence* $\equiv$ in our correctness criterion in Definition 5.1:

Definition 5.3 (Semantic Equivalence). Two variation diffs D_1 and D_2 are semantically equivalent, written $D_1 \equiv D_2$, iff their projections are equal, $\forall t \in \{b, a\} : project(D_1, t) = project(D_2, t)$.

Equality of projections is defined by equality of variation trees:

Definition 5.4 (Equality of Variation Trees). Two variation trees $T_i = (V_i, E_i, r_i, \tau_i, l_i), i \in \{1, 2\}$ are equal, $T_1 = T_2$, iff $V_1 = V_2$, $E_1 = E_2$, $r_1 = r_2$, and types τ_i and labels l_i are point-wise equal: $\forall\, v \in V_1 : (\tau_1(v) = \tau_2(v)) \land (l_1(v) = l_2(v))$.

As expected, equality of two variation diffs implies their semantic equivalence:

Definition 5.5 (Equality of Variation Diffs). Two variation diffs $D_i = (V_i, E_i, r_i, \tau_i, l_i, \Delta_i), i \in \{1, 2\}$ are equal, $D_1 = D_2$, iff $(V_1, E_1, r_1, \tau_1, l_1) = (V_2, E_2, r_2, \tau_2, l_2)$ according to Definition 5.4, and the colorings are point-wise equal: $\forall\, x \in V_1 \cup E_1 : \Delta_1(x) = \Delta_2(x)$.

Corollary 5.6. *All variation diffs D_1, D_2 that are equal $D_1 = D_2$ are also semantically equivalent $D_1 \equiv D_2$.*

5.2 Generating Views

Having identified the requirements to a view on edits to variability, we now have to find a way to construct correct views of variation diffs. In fact, the correctness criterion itself gives rise to a simple but elegant definition:

$$\begin{aligned} view_{naive}(D, \rho) := diff(&view_{tree}(project(D, b), \rho), \\ &view_{tree}(project(D, a), \rho)). \end{aligned} \tag{8}$$

This definition for a viewing function arises from rearranging the terms in the correctness criterion. Consider again the commuting diagram in Figure 5. Given a variation diff D, we ought to compute a view D' on D for a relevance ρ in terms of a function $view_{diff}$. A simple way is to just walk the long way around the corners of the diagram: First, obtain the states before $T_b = project(D, b)$ and after $T_a = project(D, a)$ the edit by means of projection. Second, create the views on these trees using $view_{tree}$. Third, compute the difference of the result.

Theorem 5.7. *$view_{naive}$ is correct (cf. Definition 5.1).*

Proof. Let T_b, T_a be two variation trees and ρ a relevance. Then

$$\begin{aligned} & view_{naive}(diff(T_b, T_a), \rho) \\ \overset{\text{Eq. 8}}{=}\ & diff(view_{tree}(project(diff(T_b, T_a), b), \rho), \\ & \qquad view_{tree}(project(diff(T_b, T_a), a), \rho)) \\ \overset{\text{Def. 5.2}}{=}\ & diff(view_{tree}(T_b, \rho), \\ & \qquad view_{tree}(T_a, \rho)). \end{aligned}$$

By Corollary 5.6 syntactic equality implies semantic equivalence. Hence, all terms above are also semantically equivalent, and thus $view_{naive}$ correct. □

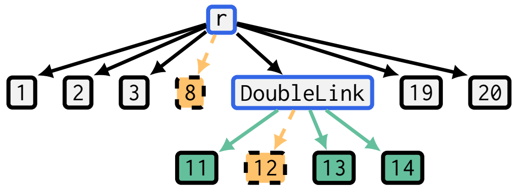

Figure 6: = *view*$_{naive}$(Figure 2, *configure*(¬Ring, *true*))

Figure 6 shows a view on the edit in Figure 2 made by Alice in our motivating example (Section 2.1). Figure 6 is the view of Charlotte who required the changes by Alice as a patch without the feature `Ring`. Listing 4 shows the corresponding concrete syntax but line numbers in Figure 6 are according to the original edit.

5.3 Optimized View Generation

An implementation of $view_{naive}$ does not seem to be efficient, hence its name. It requires two projections, two views, and, crucially, recomputing a diff despite already having a diff D as input. Given that a view only removes but never adds, we can remember whether nodes were unchanged, inserted, or deleted, instead of forgetting and recomputing this knowledge as $view_{naive}$ does. We thus develop an optimized view function that avoids recomputing a diff.

The key idea for this optimization is to reuse the functions we introduced in Section 3 for views on variation trees, but to account for times $t \in \{b, a\}$. In particular, we now track *when* a node is relevant instead of tracking just *whether* it is relevant.

To define a view on a variation diff, we have to define (1) which nodes should be in the view, (2) which edges should be in the view, and (3) the coloring $\Delta : V \cup E \to \mathcal{P}(\{b, a\})$ of nodes and edges in the view. We can compute the set of nodes and edges for a view in a similar fashion as for variation trees in Section 3. Yet, we cannot just keep the coloring as is because nodes or edges might not be relevant at all times they exist. This happened in our motivating example in Section 2.1 where Charlotte required a view on the edit shown in Listing 2 without the feature `Ring`. In the resulting view in Listing 4, a formerly unchanged line had to appear as deleted (Line 4 in Listing 4) because the line was moved from a relevant presence condition *true* to another irrelevant annotation `Ring`, that is hidden in the view defined by the relevance $configure(\neg\texttt{Ring}, true)$. This special treatment allowed Charlotte to apply the view as a patch to an older version of the software without the feature `Ring`.

To define the set of nodes and the coloring in a view on an edit, we first introduce a function which collects the times at which a node v from a variation diff D is relevant in a view:

$$tor(D, v, \rho) := \{t \in \Delta(v) \mid v \in viewnodes(project(D, t), \rho)\} \tag{9}$$

where *tor* is an abbreviation for *times of relevancy*. The times of relevancy $tor(D, v, \rho)$ of a node v in a diff D, are all times at which the node exists (i.e., $t \in \Delta(v)$) and at which it is contained in a view on the projection at that time t (i.e., $v \in viewnodes(project(D, t), \rho)$). The *viewnodes* function, defined in Equation 3, determines all nodes that should be part of a view for a variation tree. We can reuse that function to check whether a node is relevant in a projection at a certain time t. To navigate the subtree and to evaluate the relevance

ρ at a certain time t, we use the projection $\mathit{project}(D, t)$. We discuss the potential impact of this projection on implementations in Section 5.4.

Knowing at which times a node is relevant for a view on a variation diff $D = (V, E, r, \tau, l, \Delta)$, we can conclude that all nodes $v \in V$ that are relevant at least once must be in the view:

$$\mathit{viewnodes}'(D, \rho) \coloneqq \{v \in V \mid \mathit{tor}(D, v, \rho) \neq \emptyset\},$$

where $\mathit{viewnodes}'(D, \rho)$ is the set of all nodes in the view on the diff D described by the relevance ρ.

Finally, we can define our optimized operator for views on edits to variability. A view described by a relevance ρ on a variation diff $D = (V, E, r, \tau, l, \Delta)$ is given by:

$$\begin{aligned} & \mathit{view}_{\mathit{smart}}(D, \rho) \coloneqq (V', E', r, \tau, l, \Delta') \\ \textbf{where}\quad & V' \coloneqq \mathit{viewnodes}'(D, \rho), \\ & E' \coloneqq E \cap (V' \times V'), \\ & \Delta'(v) \coloneqq \mathit{tor}(D, v, \rho), \qquad \text{if } v \in V', \\ & \Delta'(e) \coloneqq \Delta'(v) \cap \Delta'(w), \qquad \text{if } e = (v, w) \in E'. \end{aligned} \tag{10}$$

This definition of views on edits is similar to the definition for views on trees in Equation 4 but extended by the coloring Δ. The set of nodes in the view is given by the function $\mathit{viewnodes}'$ defined earlier. For edges, we restrict all edges to be between nodes within the view, as we did for views on trees in Equation 4. Moreover, we have to define at which times a node or edge $x \in V \cup E$ is present in the view in terms of a coloring $\Delta : V \cup E \to \mathcal{P}(\{\mathrm{b}, \mathrm{a}\})$. For nodes, we have already defined these times in terms of the times of relevancy $\mathit{tor}(D, v, \rho)$ that denotes the set of all times at which a node v should be within the view. An edge can only exist at times at which both its nodes exist. For instance, an edge $e = (v, w)$ where v is relevant only before the edit ($\Delta'(v) = \mathit{tor}(D, v, \rho) = \{\mathrm{b}\}$) but w is always relevant ($\Delta'(v) = \mathit{tor}(D, w, \rho) = \{\mathrm{b}, \mathrm{a}\}$), can only exist before the edit since v does not exist in the view after the edit (thus, $\mathit{tor}(D, e, \rho) = \{\mathrm{b}\}$).

Theorem 5.8. *$\mathit{view}_{\mathit{smart}}$ is correct (cf. Definition 5.1).*

Proof Sketch. Let $T_{\mathrm{b}}, T_{\mathrm{a}}$ be two variation trees and ρ a relevance. Let $D = \mathit{diff}(T_{\mathrm{b}}, T_{\mathrm{a}})$. Our goal is to prove

$$\mathit{view}_{\mathit{smart}}(D, \rho) \equiv \mathit{diff}(\mathit{view}_{\mathit{tree}}(T_{\mathrm{b}}, \rho), \mathit{view}_{\mathit{tree}}(T_{\mathrm{a}}, \rho))$$

which by definition of semantic equivalence $\equiv$ (Definition 5.3) means that for all times $t \in \{\mathrm{b}, \mathrm{a}\}$, the following has to hold:

$$\begin{aligned} & \mathit{project}(\mathit{view}_{\mathit{smart}}(D, \rho), t) \\ = {} & \mathit{project}(\mathit{diff}(\mathit{view}_{\mathit{tree}}(T_{\mathrm{b}}, \rho), \mathit{view}_{\mathit{tree}}(T_{\mathrm{a}}, \rho)), t) \end{aligned}$$

The proof works by case analysis on the time t and showing that both sides of the equation simplify to the same term. The full proof is in our appendix [1]. □

5.4 Implementation

Compared to the naive view generation $\mathit{view}_{\mathit{naive}}$, defined in Equation 8, our optimized generation $\mathit{view}_{\mathit{smart}}$ does not require differencing and the separate computation of two views. Yet due to Equation 9, we still require the computation of both projections $\mathit{project}(D, t)$ to determine the relevance of a node at a certain time $t \in \{\mathrm{b}, \mathrm{a}\}$. This does not mean though that an implementation of $\mathit{view}_{\mathit{smart}}$ does indeed require to create both projections. In the end, the projection is required to (1) restrict the set of nodes to only those nodes that exist at time t, (2) compute the set of ancestors $p^+(\mathit{project}(D, t), v)$ at that time, and (3) evaluate relevance predicates ρ in the projection. All three tasks can be implemented with a single graph traversal over the variation diff D. Restricting the set of nodes (1) can be done by checking the times of existence $\Delta(v)$ of a node during the traversal. Similarly, the set of ancestors (2) in fact requires only to decide which parents to pick. The concrete relevance predicates (3) highlighted in Section 4 require the tree, and thus the projection, only to decide which parent to visit when computing presence conditions (for partial configurations or feature traces), or might not use the tree information at all (for artifact search). Thus, an implementation of $\mathit{view}_{\mathit{smart}}$ can easily be optimized to avoid full projections and instead use only the required information of parentship. For our formalization, we decided to use projections though to better express the intent of our definitions and to simplify proofs. In our implementation for the upcoming feasibility study, we implemented $\mathit{view}_{\mathit{smart}}$ with two graph traversals: One traversal to compute times of relevance, and another traversal to create a copy of the subgraph that is the view.

Having defined views for edits to variability formally, we now turn to investigating their feasibility and potential in practice in the next section. Moreover, we are interested in finding whether the optimized definition of views for edits $\mathit{view}_{\mathit{smart}}$ is indeed faster than the naive view generation $\mathit{view}_{\mathit{naive}}$.

6 FEASIBILITY STUDY

This section investigates the feasibility of generating views on edits to variational software in practice as a proof of concept. We examine whether the provided theoretical foundations of view generation can be implemented, automated, and executed within reasonable time on real-world edits. Feasibility in practice is a necessary prerequisite to study benefits for developers in the future as motivated in Section 2.1. Moreover, it is yet unknown whether our optimized view generation (Equation 10) indeed yields speedups compared to the naive one (Equation 8). Thus, we conduct an empirical feasibility study for answering the following two research questions:

RQ1 Is it feasible to generate views on edits to real-world software that implements variability via conditional compilation?

RQ2 How does the performance of the naive view generation compare to the optimized generation?

6.1 Experiment Setup

For our study, we generate views for each edit in the version history of a given software product line. We employ *DiffDetective* [14], a framework for analyzing patches and commits in the Git history of variational software implemented with the C preprocessor.

We divide each commit in the git history's default branch into one patch per edited file. Thus, for each commit, we obtain a set of text-based diffs, which we refer to as patches, each editing a different file. We discard all patches that (1) insert, delete, or move an entire file because views on these patches would actually correspond to views on source code and not on edits, or (2) do not edit a source code file (i.e., `.h`, `.hpp`, `.c`, or `.cpp`). We parse each remaining patch into a variation diff. Thereby, we collapse subsequent lines of code below the same annotation and with the same type of edit (i.e., inserted, deleted, or unchanged) to a single node.

For each variation diff, we randomly generate one relevance predicate per type introduced in Section 4:

Partial Configuration: We generate a relevance predicate that hides at least one subtree in its view. Therefore, we collect the presence conditions of all edited `artifact` nodes in the variation diff (i.e., for all artifacts that are not classified as *Untouched* [14]), for all times the node exists at. Then, we negate each presence condition such that using it as a relevance has the effect of hiding a subtree, and thus the view has a visible effect. We pick one formula randomly that is satisfiable (to not generate empty views). We then generate a relevance predicate using *configure* without a feature model (Equation 5) because feature models are unavailable for the largest parts of our dataset's histories.

Feature Traces: We collect all variable names occurring in annotation nodes, pick one at random, and create a relevance predicate via *trace* (Equation 6).

Artifact Search: We collect the lines of code within the diff, pick one at random, and plug it into *search* (Equation 7).

If a relevance predicate cannot be generated (e.g., because there is no satisfiable negated presence condition), we exclude that type of relevance predicate for the current variation diff. For each generated relevance, we run the implementations for our view algorithms $view_{naive}$ and $view_{smart}$ and measure their runtime.

For satisfiability solving, DiffDetective uses Sat4j [43], and additionally the Tseytin transformation [41, 81] for larger queries. We extended DiffDetective to decide satisfiability of small formulas (#literals<15) based on a transformation to disjunctive normal form.

The study runs on an Ubuntu 20.04.3 LTS 64-bit system and an Intel® Xeon® E5-260v3 CPU with 2.40Ghz. We process parts of the history in parallel on the system's 32 threads. To avoid impact of other processes, the machine did not execute other tasks in parallel.

Our replication package is available online [2].

6.2 Datasets

We run our experiment on each repository in the dataset collected by Liebig et al. [46] and extended by us [14]. The dataset consists of 44 open-source software repositories, covering about 30 different domains. Each software implements variability with the C preprocessor. Among others, the dataset includes widely studied subjects such as the Linux Kernel [38, 40, 52, 62, 77], Busybox [28, 38, 47, 63], and Marlin [55, 77]. For reproducibility, we use the dedicated forks provided by the DiffDetective replication package that mirror each repository in the state they were at February 2, 2022 [13].

6.3 Results and Discussion

Within about three hours, we generated almost 10 million views for about 5 million processed patches from 1,710,725 commits, in total. 8,391 patches could not be parsed due to syntactically invalid preprocessor directives (e.g., an `#endif` without `#if`). 452 patches could not be parsed due to ill-formed text-based diffs.

Table 1 presents the obtained results. It describes per view algorithm (columns $view_{naive}$ and $view_{smart}$), how many views of each type (first column) could be generated within a certain time slot

Table 1: Amount of views that could be generated within a given time slot, grouped by view type.

Type	run time is ≤	$view_{naive}$ #	$view_{naive}$ %	$view_{smart}$ #	$view_{smart}$ %
Partial Configuration	1 ms	1,121,853	70.75	1,434,649	90.48
	10 ms	358,858	22.63	137,297	8.66
	1 s	103,416	6.52	12,649	0.80
	1 min	1,456	0.09	1,001	0.06
	10 min	90	0.01	77	< 0.01
	1 h	1	< 0.01	1	< 0.01
Artifact Search	1 ms	3,812,454	77.39	4,685,940	95.13
	10 ms	1,024,747	20.80	226,123	4.59
	1 s	88,836	1.80	13,985	0.28
	1 min	11	< 0.01	0	0
Feature Traces	1 ms	2,910,342	86.60	3,290,418	97.91
	10 ms	423,344	12.60	67,631	2.01
	1 s	26,906	0.80	2,628	0.08
	1 min	73	< 0.01	0	0
	10 min	12	< 0.01	0	0

(second column), both in absolute (#) and relative numbers (%). The value for a time slot denotes that a view generation required at most this amount of time but more than the previous time slot (or 0 s in case of the first slot). For example, the time slot of 10 ms contains all view generations that required at most 10 ms but more than 1 ms (the previous time slot).

We find that generating views required less than a millisecond for 70.75 %, 77.39 %, and 86.6 % of the views per type with the naive algorithm, and more than 90 % of the views with the optimized algorithm. The median run time is 1 ms for $view_{naive}$ and a value less than 1 ms for $view_{smart}$. The reported value is 0 ms because the time measurement's resolution is limited to full milliseconds. The three views that took the longest for both algorithms were partial configurations for `src/HAL/HAL_DUE/usb/uotghs_device_due.c` in commits 02bbc511, 99ecdf59, and c0e917ea from the Marlin repository, which required about 9 – 10.6 minutes (the slowest one being Row 6 in Table 1). This file contains about 67 annotations, including two large formulas in disjunctive normal form that we suspect to cause blowups in satisfiability solving upon partial configuration. We suspect such outliers for larger changes to be reasonable. In fact, about 99.9 % of views required a second or less.

RQ1 Generating views on edits to software product lines based on conditional compilation is feasible in practice.

To determine the performance improvement of $view_{smart}$ over $view_{naive}$, we test whether there is a statistically significant difference in run times. The difference between the values of $view_{smart}$ and $view_{naive}$ is non-normally distributed. Thus, we use the Wilcoxon Signed-Rank Test with a level of significance of $\alpha = 0.005$. In particular, for all but the first time slot of each group in Table 1 (i.e., 10 ms, 1 s, etc.), we perform the test on each run time of the optimized algorithm in that slot compared to the time the naive algorithm required. We find that both distributions are significantly different (i.e., for all time slots $p\text{-}value < 10^{-12} < \alpha$) and that the mean value of the optimized algorithm is always lower than the mean value of the naive algorithm (i.e., differences range from

0.6 ms up to 985.6 ms). For more complex instances (i.e., views that required a second or longer), the naive generation was slower by a factor of about 159, 1,612, and 37 on average for the view types partial configuration, and feature traces, artifact search, respectively. We hypothesize that the performance improvement mainly results from avoiding the extra differencing step.

RQ2 On average, the optimized generation is faster to compute for views for which the naive algorithm requires one second or more.

6.4 Threats to Validity

By using DiffDetective and a similar experiment setup, our study inherits the same threats to validity as present in our earlier edit classification, which we discussed in detail [14].

The implementation of the naive algorithm (Section 5.2) depends on the chosen diff operator which in turn influences our results. We chose a line-based text diffing based on Myers algorithm, that is also used for parsing in DiffDetective because it is comparably fast in our experience. A detailed comparison of diffing algorithms can be found elsewhere [20, 25, 60] and is out of scope here.

We did not consider feature models in our study which likely yields faster runtimes. The majority of projects in our dataset do not contain explicit feature models, and recovering a feature model is a project-specific and complex task which constitutes its own area of research [32, 66]. We chose our dataset for more realistic and large-scale data, and consider investigating the impact of feature models on view generation as potential future work.

7 RELATED WORK

Related work to views on edits to variational software falls into two categories. Variation control systems reduce the complexity of editing variational systems via views on the system. Commit untangling and slicing reduce commits to identify single characteristics.

7.1 Views to Variability

Variation control systems [3, 48, 50], such as ECCO [23, 49] and SuperMod [67], employ concepts of version control systems to reduce the complexity of editing variational software. In both and similar systems [5, 77], developers edit a view on the software, obtained by checking out a (partially) configured variant, and committing the changes against a partial configuration. The superimposition of all variants is hidden from the user and is updated upon commit. While variation control systems allow for navigating the version history and to edit single views, they do not allow to perform views on the changes themselves. Similarly, Kästner et al. support views in the software product line editor CIDE [31, 34] both via partial configuration and feature traces but describe views only informally and do not consider views on edits. *Feature revisions* [59, 69] identify changes made to a specific feature, but in contrast to our formalization, are restricted to a single feature per view.

Walkingshaw and Ostermann formalized the workflow of variation control systems [85] based on the choice calculus, a formal and minimal language for variability annotations [19, 84]. Thereby, views on variability are formalized as a partial configuration upon checkout (*get*). In this work, we generalized producing views on variability to other types of views, including feature traces and artifact search. The restriction to partial configurations is necessary though to enable applying an edit to a view (i.e., a view on an edit, cf. Section 5.1) to the underlying complex system (*put*) [85]. Extending the whole workflow of variation control systems to the other types of views also requires a specialized commit (*put*) operator for each view type, which would be interesting future work.

7.2 Commit Untangling

Commit untangling, or commit slicing, splits commits that address multiple concerns (e.g., a refactoring and a bug fix) into sets of changes addressing a single concern each [15, 45, 58, 61, 73, 87]. Techniques for commit untangling can roughly be divided into two categories [45], with a third category being currently established. First, techniques that leverage ideas from mining software repositories usually consider repository properties such as overlaps in the change sets of commits [36, 37], package and file distances of changes [27], artifact co-changes in the history [27], or code dependencies [27, 39, 79]. Second, techniques based on static code analyses detect source code relationships at varying levels of detail, such as define-use chains [7], data flow and control flow [57, 61], relational links between patches [73], or refactorings [75]. Third, recent techniques [15, 45] combine static code analysis with machine learning to extract meaningful change representations and to automatically separate change sets.

Both our views on edits and commit untangling aim to identify changes to distinct concerns that were made in an edit to multiple concerns at once. In commit untangling, these concerns align with the source code. Our views on edits cover variational concerns such as configuration or feature tracing (cf. Section 4). While commit untangling operates on project metadata and the used programming languages, our views operate on a meta-language that annotates these programming languages. Thus, commit untangling and our views serve orthogonal concerns and could be applied in parallel.

8 CONCLUSION

We developed the theoretical foundations for views on edits to variational software, and observed that creating views on edits can be reduced to creating usual views on single revisions of the variational software. Based on this observation, we formulated a correctness criterion that should be satisfied by any view. We proposed two algorithms for view construction and proved their correctness. First, a naive and simple algorithm directly emerged from the correctness criterion and is suitable for use in proofs and formal reasoning. Second, a runtime-optimized algorithm avoids redundant computations. We empirically confirmed the feasibility of generating views on edits to variational software by applying our algorithms to the version histories of 44 open source and real-world software product lines.

As we confirmed that generating views on edits to variational software is feasible even in large projects, future work may evaluate the practicality of views on edits for developers in subsequent experiments, such as patch backporting, or dedicated user studies. Upon view generation, also annotations may be simplified according to the relevance predicate.

ACKNOWLEDGMENTS

We thank Chico Sundermann and Lukas Bormann for helpful discussions, and Tim Nolle for testing our replication package. 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] 2023. Appendix. https://github.com/VariantSync/DiffDetective/raw/splc23-views/appendix/appendix-splc23-views.pdf; The appendix is also part of our replication package [2].

[2] 2023. Replication Package. https://github.com/VariantSync/DiffDetective/tree/splc23-views/replication/splc23-views. https://doi.org/10.5281/zenodo.8027920

[3] Sofia Ananieva, Sandra Greiner, Timo Kehrer, Jacob Krüger, Thomas Kühn, Lukas Linsbauer, Sten Grüner, Anne Koziolek, Henrik Lönn, S. Ramesh, and Ralf H. Reussner. 2022. A Conceptual Model for Unifying Variability in Space and Time: Rationale, Validation, and Illustrative Applications. *Empirical Software Engineering (EMSE)* 27, 5 (2022), 101.

[4] Sofia Ananieva, Sandra Greiner, Jacob Krueger, Lukas Linsbauer, Sten Gruener, Timo Kehrer, Thomas Kuehn, Christoph Seidl, and Ralf Reussner. 2022. Unified Operations for Variability in Space and Time. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 7, 10 pages.

[5] Sofia Ananieva, Thomas Kühn, and Ralf Reussner. 2022. Preserving Consistency of Interrelated Models during View-Based Evolution of Variable Systems. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 148–163.

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

[7] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. 2015. Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, Vol. 1. IEEE, 134–144.

[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] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 16–25.

[10] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Feature-to-Code Mapping in Two Large Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, Vol. 6287. Springer, 498–499.

[11] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. 1993. The Concept Assignment Problem in Program Understanding. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 482–498.

[12] 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.

[13] Paul Maximilian Bittner, Christof Tinnes, Alexander Schultheiß, Sören Viegener, Timo Kehrer, and Thomas Thüm. 2022. *Appendix and Replication Package for Article: Classifying Edits to Variability in Source Code.*

[14] 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.

[15] Siyu Chen, Shengbin Xu, Yuan Yao, and Feng Xu. 2022. Untangling Composite Commits by Attributed Graph Clustering. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware*. ACM, 117–126.

[16] Reidar Conradi and Bernhard Westfechtel. 1998. Version Models for Software Configuration Management. *ACM Computing Surveys (CSUR)* 30, 2 (1998), 232–282.

[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] 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.

[20] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. 313–324.

[21] 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 *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 1174–1185.

[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] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 665–668.

[24] Sandra Greiner and Bernhard Westfechtel. 2021. On Preserving Variability Consistency in Multiple Models. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 7, 10 pages.

[25] Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *Proc. Working Conf. on Reverse Engineering (WCRE)*. 279–288.

[26] Florian Heidenreich, Jan Kopcsek, and Christian Wende. 2008. FeatureMapper: Mapping Features to Models. In *Companion Int'l Conf. on Software Engineering (ICSEC)*. ACM, 943–944. Informal demonstration paper.

[27] Kim Herzig, Sascha Just, and Andreas Zeller. 2016. The Impact of Tangled Code Changes on Defect Prediction Models. *Empirical Software Engineering (EMSE)* 21 (2016), 303–336.

[28] Tobias Heß, Chico Sundermann, and Thomas Thüm. 2021. On the Scalability of Building Binary Decision Diagrams for Current Feature Models. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 131–135.

[29] 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.

[30] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report CMU/SEI-90-TR-21. Software Engineering Institute.

[31] Christian Kästner. 2010. *Virtual Separation of Concerns: Toward Preprocessors 2.0.* Ph. D. Dissertation. University of Magdeburg.

[32] Christian Kästner. 2017. *Differential Testing for Variational Analyses: Experience from Developing KConfigReader.* Technical Report arXiv:1706.09357. Cornell University Library.

[33] 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.

[34] Christian Kästner, Salvador Trujillo, and Sven Apel. 2008. Visualizing Software Product Line Variabilities in Source Code. In *Proc. Int'l Workshop on Visualisation in Software Product Line Engineering (ViSPLE)*. 303–313.

[35] Timo Kehrer, Udo Kelter, Pit Pietsch, and Maik Schmidt. 2012. Adaptability of Model Comparison Tools. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 306–309.

[36] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2014. Hey! Are You Committing Tangled Changes?. In *Proc. Int'l Conf. on Program Comprehension (ICPC)*. ACM, 262–265.

[37] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2016. Splitting Commits via Past Code Changes. In *Proc. Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 129–136.

[38] 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 *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 291–302.

[39] Patrick Kreutzer, Georg Dotzler, Matthias Ring, Bjoern M. Eskofier, and Michael Philippsen. 2016. Automatic Clustering of Code Changes. In *Proc. Working Conf. on Mining Software Repositories (MSR)*. ACM, 61–72.

[40] 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.

[41] 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 *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 110:1–110:13.

[42] 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.

[43] Daniel Le Berre and Anne Parrain. 2010. The Sat4j Library, Release 2.2. *J. Satisfiability, Boolean Modeling and Computation* 7, 2-3 (2010), 59–64.

[44] Yi Li, Julia Rubin, and Marsha Chechik. 2023. Semantic History Slicing. In *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines*. Springer, 53–77.

[45] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. UTANGO: Untangling Commits with Context-Aware, Graph-Based, Code Change Clustering Learning Model. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 221–232.

[46] 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.

[47] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 81–91.

[48] 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.

[49] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability Extraction and Modeling for Product Variants. *Software and System Modeling (SoSyM)* 16, 4 (2017), 1179–1199.

[50] Lukas Linsbauer, Felix Schwägerl, Thorsten Berger, and Paul Grünbacher. 2021. Concepts of Variation Control Systems. *J. Systems and Software (JSS)* 171 (2021), 110796.

[51] 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.

[52] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux Kernel Variability Model. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 136–150.

[53] 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.

[54] 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 *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2–15.

[55] 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 *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, Article 14, 11 pages.

[56] 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 *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 289–301.

[57] Ward Muylaert and Coen De Roover. 2018. Untangling Composite Commits Using Program Slicing. In *Proc. Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM)*. IEEE, 193–202.

[58] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. 2019. Graph-Based Mining of in-the-Wild, Fine-Grained, Semantic Code Change Patterns. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 819–830.

[59] Michael Nieke, Gil Engel, and Christoph Seidl. 2017. DarwinSPL: An Integrated Tool Suite for Modeling Evolving Context-aware Software Product Lines. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 92–99.

[60] Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto. 2020. How Different are Different Diff Algorithms in Git? *Empirical Software Engineering (EMSE)* 25, 1 (2020), 790–823.

[61] Profir-Petru Pârtachi, Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2020. Flexeme: Untangling Commits Using Lexical Flows. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 63–74.

[62] 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. *Empirical Software Engineering (EMSE)* 21, 4 (2016).

[63] Tobias Pett, Sebastian Krieter, Tobias Runge, Thomas Thüm, Malte Lochau, and Ina Schaefer. 2021. Stability of Product-Line Sampling in Continuous Integration. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 18, 9 pages.

[64] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[65] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering: Product Lines, Languages, and Conceptual Models*, Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and Jorn Bettin (Eds.). Springer, 29–58.

[66] Alexander Schultheiß, Paul Maximilian Bittner, Sascha El-Sharkawy, Thomas Thüm, and Timo Kehrer. 2022. Simulating the Evolution of Clone-and-Own Projects with VEVOS. In *Proc. Int'l Conf. on Evaluation Assessment in Software Engineering (EASE)*. ACM, 231–236.

[67] Felix Schwägerl and Bernhard Westfechtel. 2016. SuperMod: Tool Support for Collaborative Filtered Model-Driven Software Product Line Engineering. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 822–827.

[68] Felix Schwägerl and Bernhard Westfechtel. 2019. Integrated Revision and Variation Control for Evolving Model-Driven Software Product Lines. *Software and System Modeling (SoSyM)* 18, 6 (2019), 3373–3420.

[69] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. DeltaEcore - A Model-Based Delta Language Generation Framework. In *Proc. Modellierung*. Gesellschaft für Informatik, 81–96.

[70] Francisco Servant and James A. Jones. 2012. History Slicing: Assisting Code-Evolution Tasks. In *Proc. Int'l Symposium on Foundations of Software Engineering (FSE)*. ACM, Article 43, 11 pages.

[71] Francisco Servant and James A. Jones. 2013. Chronos: Visualizing Slices of Source-Code History. In *Proc. Working Conf. on Software Visualization (VISSOFT)*. IEEE, 1–4.

[72] Ridwan Shariffdeen, Xiang Gao, Gregory J. Duck, Shin Hwei Tan, Julia Lawall, and Abhik Roychoudhury. 2021. Automated Patch Backporting in Linux (Experience Paper). In *Proc. Int'l Symposium on Software Testing and Analysis (ISSTA)*. ACM, 633–645.

[73] Bo Shen, Wei Zhang, Christian Kästner, Haiyan Zhao, Zhao Wei, Guangtai Liang, and Zhi Jin. 2021. SmartCommit: A Graph-Based Interactive Assistant for Activity-Oriented Commits. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 379–390.

[74] Janet Siegmund. 2016. Program Comprehension: Past, Present, and Future. In *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 13–20.

[75] Sarocha Sothornprapakorn, Shinpei Hayashi, and Motoshi Saeki. 2018. Visualizing a Tangled Change for Supporting Its Decomposition and Commit Construction. In *Proc. on Computer Software and Applications Conf. (COMPSAC)*, Vol. 01. IEEE, 74–79.

[76] Christopher S. Strachey. 2000. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation* 13, 1/2 (2000), 11–49.

[77] 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.

[78] Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2020. Evaluating #SAT Solvers on Industrial Feature Models. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 3, 9 pages.

[79] Yida Tao and Sunghun Kim. [n. d.]. Partitioning Composite Code Changes to Facilitate Code Review. In *Proc. Working Conf. on Mining Software Repositories (MSR)*. IEEE, 180–190.

[80] 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 *Proc. Int'l Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution)*. ACM, 57–64.

[81] Grigori S. Tseytin. 1983. *On the Complexity of Derivation in Propositional Calculus*. Springer, 466–483.

[82] Sören Viegener. 2021. *Empirical Evaluation of Feature Trace Recording on the Edit History of Marlin*. Bachelor's Thesis. University of Ulm.

[83] Anneliese von Mayrhauser, A. Marie Vans, and Adele E. Howe. 1997. Program Understanding Behaviour During Enhancement of Large-Scale Software. *J. Software: Evolution and Process* 9, 5 (1997), 299–327.

[84] Eric Walkingshaw. 2013. *The Choice Calculus: A Formal Language of Variation*. Ph. D. Dissertation. Oregon State University.

[85] 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.

[86] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2013. How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study. *J. Software: Evolution and Process* 25, 11 (2013), 1193–1224. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1593

[87] Min Wang, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2020. CoRA: Decomposing and Describing Tangled Code Changes for Reviewer. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 1050–1061.

[88] Chenguang Zhu, Yi Li, Julia Rubin, and Marsha Chechik. 2020. GenSlice: Generalized Semantic History Slicing. In *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 81–91.