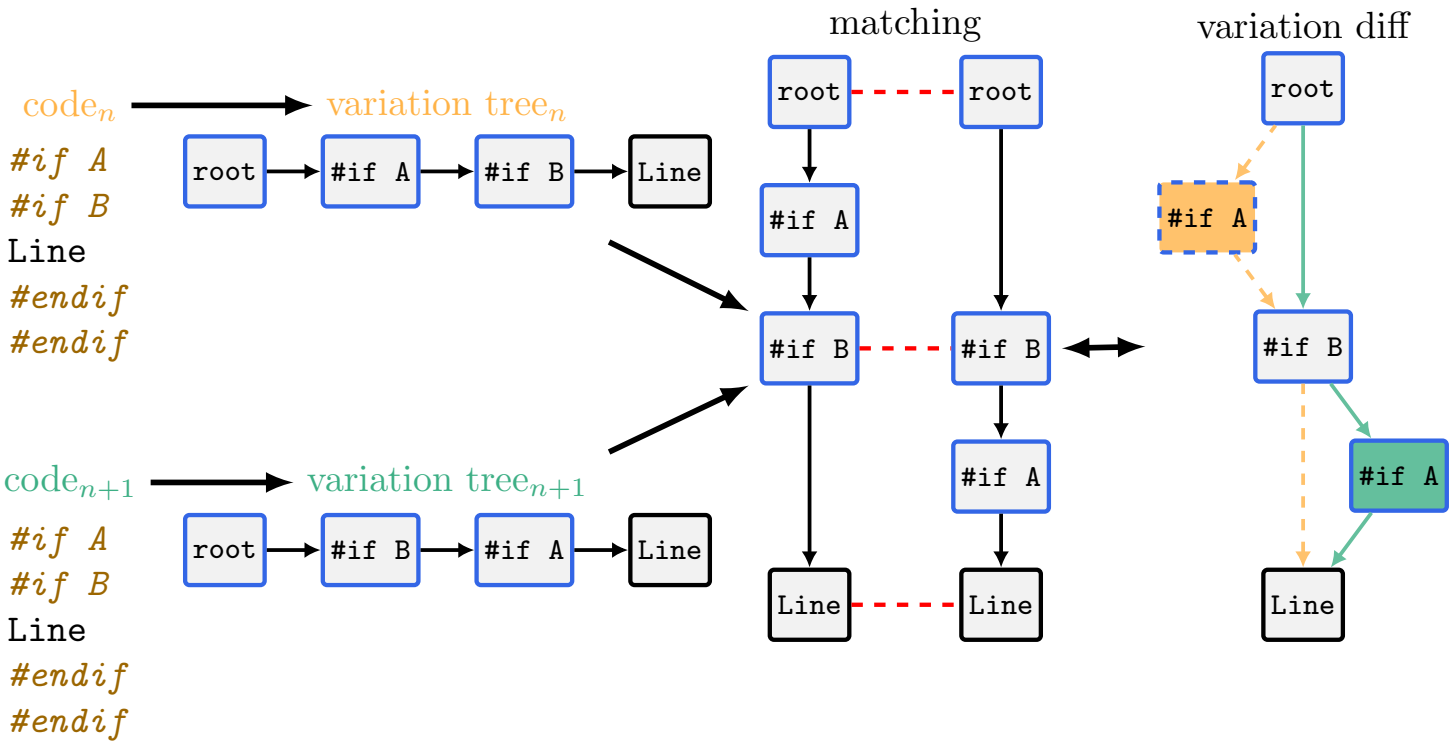
Software Engineering
Programming Languages

universität ulm



Constructing Variation Diffs Using Tree Diffing Algorithms

Benjamin Moosherr

Bachelor's Thesis

Institute of Software Engineering and Programming Languages

Benjamin Moosherr:
Constructing Variation Diffs Using Tree Diffing Algorithms

Advisors:

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

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

Bachelor's Thesis, University of Ulm, 2023.

Abstract

Changes to the artifacts, such as source code, of software product lines often affect a multitude of variants. Variation diffs are a complete model for describing and analyzing changes to artifacts and all their feature-to-code-mappings. While variation diffs offer flexibility in their granularity and edit representation, this flexibility is currently not exhausted due to short-comings of the state-of-the-art construction algorithm. We contribute a variation diff construction algorithm based on tree diffing which reduces the construction of variation diffs to finding a matching between two variation trees. Moreover, we prove that a variation diff for a concrete edit is uniquely determined by such a matching. We discuss quality dimensions of variation diffs and metrics for measuring relevant quality differences. Besides repurposing metrics used for line and tree differs, we present an application-specific metric in the domain of classifying edits to software product lines. In a case study with the software product lines Vim, Busybox, and Marlin, we compare the quality of variation diffs constructed using the matching of a line differ, a tree differ, and a hybrid approach. Thereby, we validate our theoretical results and measure the performance of each algorithm to assess their feasibility in practice. We find that the performance and quality of constructing variation diffs using a tree matcher is worse than using the state-of-the-art algorithm. However, our construction algorithm improves the quality of variation diffs by combining a line and tree matcher into a hybrid matcher.

Contents

List of Figures

List of Tables

1. Introduction

A software product line implements a collection of related software variants while sharing implementation artifacts, for instance source code lines, between these variants [2]. To identify a variant, it is often associated with a set of features implemented in that variant [2, 20]. For example, the Linux kernel [43] implements support for different hardware as a set of features. Some subsets of these features can be chosen to produce a Linux variant. However, features may have complicated relations to each other and the set of all variants may be very large [40]. This can pose challenging problems when working with software product lines.

There is one particular problem which arises during the initial development, the evolution and the maintenance of a software product line [39]: Changes to the implementation artifacts of a software product line can cause subtle defects in feature models and feature mappings [40]. For example, users and developers of the Linux kernel requested runtime-loadable kernel modules [42]. This feature request not only required changes to the feature model of the Linux kernel but also changes to all implementation artifacts to be loaded at runtime and the addition of new artifacts implementing the module loading. Both, changes to the feature model and the implementation artifacts, especially in the presence of feature mappings, can be hard to understand.

For developers, it is important to understand the implications of changes made to the implementation artifacts of a software product line to reduce the amount of errors. Formally, a software product line can be modelled by a variation tree which associates each artifact with a presence condition stating which variants include a given implementation artifact [6]. A useful tool for analyzing changes to a variation tree is the variation diff [6], an example of which can be as seen in Figure 1.1a. A variation diff encodes two variation trees, one before the change and one after the change, by specifying, for each node and edge, whether it was inserted (green), deleted (orange) or unchanged (black). Unfortunately, the construction of such variation diffs is ambiguous [6], demonstrated by Figure 1.1b which encodes the same change as Figure 1.1a. Indeed, the quality[1] of the resulting variation diff depends on the construction algorithm. These quality differences

[1] Metrics for measuring the quality of variation diffs are discussed in Chapter 4.

(a) Example variation diff, changing the order of the mapping nodes `#ifdef A` and `#ifdef B`

(b) Variation diff equivalent to Figure 1.1a, but the movement of the `#ifdef B` node was detected

Figure 1.1: Comparison of two equivalent variation diffs

possibly influence the usefulness of variation diffs for developers who want to analyze changes.

The current approach for constructing variation diffs uses a line-based diffing algorithm [6, 44] (e.g., Myers algorithm [34]) to detect inserted, deleted, and unmodified lines. This construction of variation diffs does not incorporate the detection of moved artifacts, introducing mental overhead for developers analyzing common changes to source code [13]. For example, in Figure 1.1b the move of the `#ifdef B` node was detected whereas in Figure 1.1a it was interpreted as a separate delete and insert operation. Moreover, Bittner et al. [6] laid the foundation of handling more fine grained changes in variation trees without implementing the possibility for such granularity when constructing their variation diffs. However, there are alternative diffing algorithms which can, for example, detect moved lines [8, 22] or operate on (syntax) trees instead of lines [9, 16, 38]. Notably, variation trees may be diffed using such tree diffing algorithms. Ideally, a variation diff construction algorithm could detect moved artifacts while being able to process variation trees with different levels of granularity.

In this thesis, we present a novel algorithm for constructing variation diffs using matchings which is able to process abstract syntax trees with different granularities. In contrast to using line-based diffing, as seen with solid arrows in Figure 1.2, we construct variation trees before and after a change and apply part of a tree diffing algorithm, marked by dashed arrows in Figure 1.2. We formalize this process and prove the equivalence between variation diffs and matchings between two variation trees. For comparing different variation diffs according to their usefulness for developers and analyses based on variation diffs, we discuss metrics for variation diffs. Then, we empirically evaluate different matching algorithms, including Viegener's original algorithm, based on our metrics using the implementation we provide in DiffDetective. In summary, our contributions are:

Contruction algorithm: We present a novel variation diff construction algorithm.

Implementation: We provide an implementation of the presented algorithm in DiffDetective.

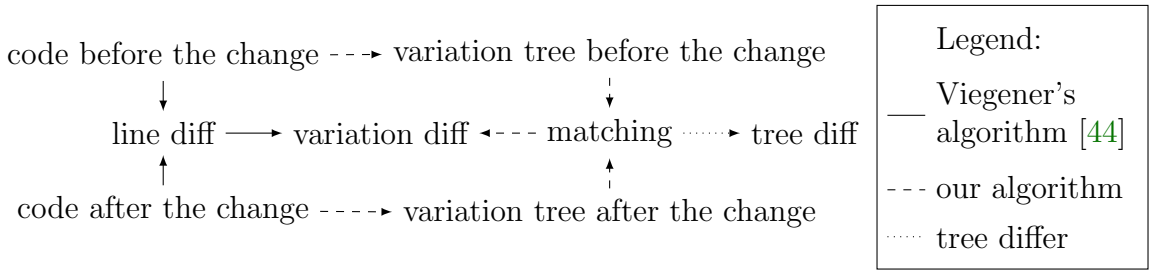


Figure 1.2: Dataflow during the construction of variation diffs

Conceptual: We formally analyse the relationship between variation diffs and matchings.

Metrics: We discuss possible metrics for comparing variation diffs.

Evaluation: We evaluate the performance and compare different matching strategies using the discussed metrics.

The rest of this thesis is structured in the following way: We start by providing some required background knowledge in Chapter 2. In Chapter 3 we explain how tree diffing algorithms can be applied to variation trees and analyze their relationship formally. Next, we discuss quality metrics useable for comparing variation diffs in Chapter 4. In Chapter 5 we evaluate our concepts and implementation using the discussed quality metrics and a benchmark. Finally, we discuss related work Chapter 6 in and we provide our conclusion in Chapter 7.

2. Background

2.1 C Preprocessor

The C preprocessor is a textual macro processor. It originated in the C programming language and was first standardized by the American National Standards Institute (ANSI) in 1989 [47] and by the International Organization for Standardization (ISO) in 1990 [48]. The C preprocessor is also used for some other languages like C++ and Haskell, although its operation is sometimes slightly modified. The current international standards of the C preprocessor are C17 [49] and C++20 [1] which differ slightly but are kept mostly in sync.

The most relevant processing performed by the C preprocessor is interpreting lines containing preprocessor directives. All other lines are considered raw text lines[1] and are passed to the next translation phase, most of the time a traditional actual parser. For the purpose of this thesis, we ignore the following translation phases as we do not want to parse the C code.

The syntax of a preprocessor directive consists of a `#` as the first non-blank character followed by a white space delimited command and its optional argument in the rest of the line. An example can be seen in Figure 2.1. It contains two nested if-chains consisting of `#if`, `#elif`, `#else` and `#endif` preprocessor directives, in that order. The `#elif` directive can occur an arbitrary number of times and the `#else` directive is optional. Furthermore, the `#if` and `#elif` preprocessor directives require one argument which is a C-like expression, called the condition of the branch. A branch is the file content between two preprocessor directives of the same if-chain without another preprocessor directive of the same if-chain in between. For example, in Figure 2.1 a branch consists of consecutive lines with at least the same indentation. This indentation scheme automatically depicts the number of nested if-chains at a specific line which is equivalent to the indentations level.

The semantic of if-chains follows well known principles: Each branch of an if-chain is examined in order from top to bottom of the file. The first branch whose condition

[1] The C preprocessor actually transforms all lines by applying text replacements but this is irrelevant for this thesis and therefore not explained.

```
1
#if FEATURE_A && FEATURE_B
    2
    #if __STDC_VERSION__ >= 201710L
        Using_C17_or_newer
    #else
        Using_C11_or_older
    #endif
#elif FEATURE_B
    5
#else
    6
#endif
```

Figure 2.1: Example of C preprocessor code for implementing different features

evaluates to true, is considered active. If a branch starts with `#else` its condition is considered as always true. If an active branch was found or all branches where considered, the lines of all non-active branches are discarded and not processed any further. Then the lines of the active branch are evaluated recursively, if there are any. As mentioned above, the text lines which remain after all preprocessor directives have been executed are the result of the C preprocessor.

There are two steps in the evaluation of the C-like conditions. First, all identifiers are replaced by there macro value or `0` if they do not have a value. This value can be set when invoking the compiler or during compilation by other preprocessor directives. The result will be an arithmetic expression of the language C with only constants as arguments. Second, the arithmetic expression is evaluated. Common operators used by C preprocessor conditions are logical `And` denoted by `&&`, logical `Or` denoted by `||`, and logical `Negation` denoted by `!`. For example, if `FEATURE_A` and `FEATURE_B` are defined when executing a C11 compilation[2] for Figure 2.1, then the first branch of the outer and the second branch of the inner if-chain are the result of the C preprocessor (black). All other lines (grey) are discarded or are preprocessor directives (brown) which are not included in the result either.

2.2 Software Product Lines

A software product line implements a collection of related software variants. In contrast to unrelated software variants, these variants have to share implementation artifacts [2], for instance source code lines. For identification, a variant is associated with a set of features implemented in that variant [2, 20]. For example, the Linux kernel [43] implements support for different hardware as a set of features. Some subsets of these features can be chosen to produce a Linux variant. Hence, the Linux kernel is considered a software product line.

To share artifacts, each artifact is annotated with a propositional formula called the presence condition of an artifact [6]. This presence condition specifies exactly which variants

[2]The compiler automatically defines macros like `__STDC_VERSION__` to expose supported compiler features to the source code.

contain the annotated artifact. A presence condition is evaluated by associating each variable of the propositional formula with a feature and assigning a truth value based on the presence of that feature in a variant. For easier maintenance, presence conditions are often split into a sequence of feature mappings represented as a tree.

One implementation technology for associating each artifact with a presence is the C preprocessor. Thus, Figure 2.1 represents the artifacts of a software product line. In the case of the C processor, artifacts are source code lines. In Figure 2.1, the artifacts are, 1, 2, `Using_C17_or_newer`, `Using_C11_or_older`, 5, and 6. Presence conditions are annotated by preprocessor directives and are commonly split into multiple feature mappings. For example, `Using_C17_or_newer` is annotated by the feature mappings[3] `FEATURE_A && FEATURE_B` and `__STDC_VERSION__`. However, the presence condition of `Using_C17_or_newer` is `FEATURE_A && FEATURE_B && __STDC_VERSION__`. Different variants of this example software product line can be obtained by selecting different features. For example, selecting both feature `FEATURE_A` and `FEATURE_B` describes a variants Figure 2.1. A different variation may be obtained by selecting the feature `FEATURE_A` and deselecting the feature `FEATURE_B`.

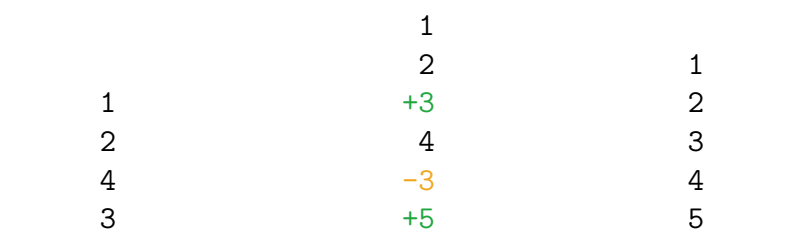
2.3 Line-Based Diffing

Line-based diffing algorithms compute the difference between the lines of two files. The result of such a computation is an edit script representing the differences as a list of edit actions performed on one file to obtain the other file. Replaying the edit actions of an edit script is often called patching. A common type of edit script are line-based diffs which consist of line deletion and insertion actions. Myer's algorithm [34] demonstrates that line-based diffs can be computed efficiently. Hence, it Myer's algorithm is used in many practical applications. For example, the default diffing algorithm of the source control management system Git is Myer's algorithm.

A common notation for line-based diffs, prefixes each line by a single character, called the diff symbol, indicating the operation which was performed on it [11]. Hereby, a + marks added lines and a - marks removed lines. Unchanged lines are prefixed by a space, although they are often omitted if there is enough context to identify the position of the added and removed lines. An example can be seen in Figure 2.2, where the original file, a patch and the resulting file are show side by side. For convenience added lines are also colored green and removed lines red.

Another type of edit script actions are line movements [22]. Line movements can explicitly state common edits performed by developers, for example reordering functions in a class. Consider Line 3 in Figure 2.2. It was moved one line upwards, but the diff does not indicate this fact directly. Instead, Line 3 is marked as deleted and reinserted after line 4. There are algorithms which can detect such line movements [8, 22], however there is no common notation for line-based diffs with line movements.

[3] For ease of understanding, we reuse the C preprocessor syntax instead of using abstract propositional formulas.



(a) Original file (b) Line base diff (c) Resulting file

Figure 2.2: Example of a line diff between two files

2.4 Mathematical Conventions

We consider only finite sets. In principle everything proven in this thesis could be generalized to at least countably infinite sets but, as we are mostly concerned with practical applications, we prioritise simpler proofs.

notation	meaning
id	The identity function $id(x) = x$
f^{-1}	The inverse of the function f
$g \circ f$	The composition of two functions f and g such that $(g \circ f)(x) = g(f(x))$ for all x
$f(A)$	A function $f : B \to C$ applied to all elements in the set $A \subseteq B$, formally $f(A) := \{f(a) : a \in A\}$
$f\|_A$	The function $f : B \to C$ with the domain restricted to $A \subseteq B$, thus $f\|_A : A \to C$
$A \cong B$	The structures A and B are isomorph. Intuitively, the objects (e.g., trees) A and B have the same structure (e.g., only the node identifiers of two trees are different, but the parent-child relationships and the node labels stay the same) and can thus be transformed into each other without loss of information and without violating the relationships (e.g., parent-child relationship) of the parts (e.g., nodes) of the objects.

3. Variation Diff Construction Using Matchings

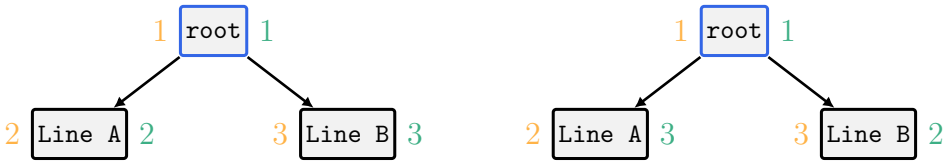
In this chapter, we present a general algorithm for constructing variation diffs which is able to take advantage of all the flexibility variation diffs offer. For example, this algorithm makes it possible to construct variation diffs with arbitrary granularity and node movements.

We present an algorithm for constructing variation diffs from matchings between variation trees and we prove the equivalence between variation diffs and matchings between variation trees. In particular, we prove that our construction algorithm is a bijection from variation diffs to matchings between variation trees. For representing all node movements correctly, we extend variation diffs, a formal model for edits to software product lines by Bittner et al. [6], with children order.

We start in Section 3.1 with the extension of variation trees and variation diffs with child ordering. Furthermore, we define the concept of semantically equivalent variation diffs and define matchings between variation trees. In Section 3.2, we present our function for constructing variation diffs from matchings between variation trees and proof its consistency and correctness. Finally, we proof that our construction is bijection in Section 3.3

3.1 Definitions

Variation trees are a formal model for artifacts of software product lines introduced by Bittner et al. [6]. They associate each artifact with a presence condition consisting of the conjunction of all feature mappings in the path from the artifact to to the root. Figure 3.2 shows two variation trees on the left side. The semantic of both variation trees in Figure 3.2 is the same and states that the artifact `A` is present in the variants containing the features `A` and `B`. Each tree node is represented by a rectangle whereas edges are represent by black arrows pointing from the parent to its child. A node with a black border, for example `Line`, is an artifact, all other nodes are of type `mapping`or `else`. The text in each node represents the label with a C preprocessor like syntax. Thus



(a) unchanged (b) order flipped

Figure 3.1: Movement of artifacts below the same parent

`#ifdef A` represents an annotation with the feature mapping `A` and a node with the label `#else` represents a node of type `else`.

Definition 3.1 (Variation tree [6])**.** *A variation tree* (V, E, r, τ, l, O) *is a rooted ordered tree with*

- *nodes* V,
- *edges* $E \subseteq V \times V$,
- *the root* $r \in V$,
- *node types* $\tau : V \to \{\texttt{artifact}, \texttt{mapping}, \texttt{else}\}$,
- *labels* $l : V \to A \cup \mathcal{P}$, *where* A *is the set of all artifacts and* $\mathcal{P}$ *the set of all propositional formulas, and*
- *an injective function* $O : V \to \mathbb{N}$ *which defines an order for the children of each node.*

The root r *must must have the type* $\tau(r) = \texttt{mapping}$ *with the label* $l(r) = \texttt{true}$. *A node* v *with* $\tau(v) = \texttt{artifact}$ *is called artifact node and must have an artifact* $a \in A$ *as label* $l(v) = a$. *Similarly, a node* v *with* $\tau(v) = \texttt{mapping}$ *is called a mapping node and must have a feature mapping, a propositional formula* $p \in \mathcal{P}$, *as label* $l(v) = p$. *In contrast, a node* v *with* $\tau(v) = \texttt{else}$ *is called an else node, but has an empty label* $l(v)$ *and must be the only else node of an if node.*

This definition of variation trees diverges from the definition given by Bittner et al. [6] in that it introduces child ordering. We require variation trees to be ordered because this ordering information is necessary to encode node movements which do not change the parent of a node in variation diffs. Similar to variation trees, we adopt the variation diff definition by Bittner et al. [6] but introduce the notion of child ordering. An example of this child ordering can be seen in Figure 3.1. The children order before the edit are indicated by the yellow numbers left of a node whereas the children order after the edit is the green numbers right of a node. The variation diff in Figure 3.1a depicts a variation diff with two unchanged artifacts. In contrast, the variation diff in Figure 3.1b depicts a variation diff where one artifact is moved from the left side to the right side of another artifact. The variation diff definition given by Bittner et al. [6] cannot distinguish these two cases because they differ only in their children order.

Variation diffs are a formal model for changes to variation trees which extend variation trees with a notion of a time before and after an edit. Thus, the semantic of a variation diff

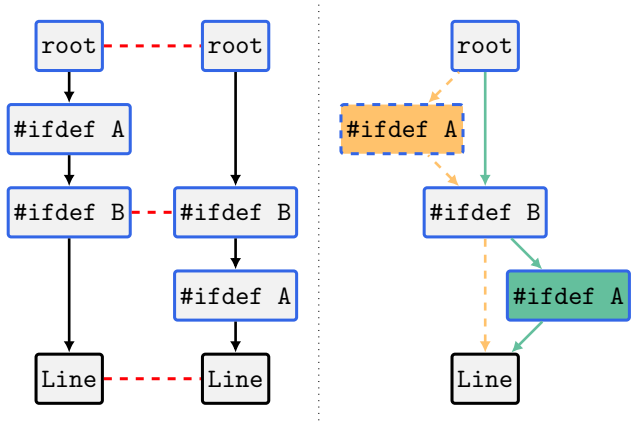


Figure 3.2: Two variation trees with a matching and the corresponding variation variation diff

D is defined by its projections $project(D, t)$ for all $t \in \{\texttt{before}, \texttt{after}\}$ which are the variation trees before ($t = \texttt{before}$) and after ($t = \texttt{after}$) the edit represented by D. An example can be seen in Figure 3.2. On the right there is a variation diff, on the left there are its projections. The variation diff represents a variation tree where an annotation with the feature mapping A was removed above and inserted below the annotation with the feature mapping B. Nodes and edges of variation diffs are drawn like variation trees (see Section 2.2) if they are unchanged. In particular, annotation nodes have a blue, artifact nodes a black border and the text of a node is the line of the C preprocessor source code which the node represents (see Section 2.1). The only exception is the artificial root node which does not have a representation in C preprocessor source code but always has to have the feature mapping `true`anyways. The time of existence is marked by the color of the edges and node interior, representing • as black in the case of edges and white in the case of nodes, - as dashed orange and + as green. The child ordering is unique in this example, so they is not depicted. In the following, we will only depict the children order if it is important to the example, otherwise we assume that the children are ordered from left to right or from top to bottom.

Our definition of variation diffs does not always uniquely specify which nodes are moved. In particular, variation diffs do not encode which children have been moved, only that they have been moved. The ambiguity can be seen in Figure 3.1b, where it is unclear whether `Line A` or `Line B` has been moved. Such information is generally encoded using edit scripts. There are algorithms for computing a minimum edit script, a well known example for tree edit scripts was presented by Chawathe et al. [9]. However, edit scripts are not unique in general.

Definition 3.2 (Variation diff [6])**.** *A variation diff* $D = (V, E, r, \tau, l, \Delta, O_{before}, O_{after})$ *is a rooted directed connected acyclic graph with*

- *nodes* V,
- *edges* $E \subseteq V \times V$,
- *the root node* $r \in V$,
- *node types* $\tau : V \to \{\texttt{artifact}, \texttt{mapping}, \texttt{else}\}$,

- *node labels* $l : V \to A \cup \mathcal{P}$*, where* A *is the set of all artifacts and* $\mathcal{P}$ *the set of all propositional formulas,*
- *the time of existence* $\Delta : V \cup E \to \{+, -, \bullet\}$ *that defines if a node or edge was added* $+$*, removed* $-$*, or unchanged* $\bullet$*, and*
- *the children before* O_{before} *and after* O_{after} *the edit being injective functions* $O_{before}, O_{after} : V \to \mathbb{N}$*.*

The projections $project(D, t)$ *for all times* $t \in \{\texttt{before}, \texttt{after}\}$ *have to be variation diffs with the same root.*

Definition 3.3 (Projection [6])**.** *The projection* $project(D, t)$ *of a variation diff* $D = (V, E, r, \tau, l, \Delta, O_{before}, O_{after})$ *at the time* $t \in \{\texttt{before}, \texttt{after}\}$ *is defined as*

$$project(D, t) := (V', E', r, \tau|_{V'}, l|_{V'}, O_t) \tag{3.1}$$

where $V' = \{v \in V : exists(\Delta(v), t)\}$*,* $E' = \{e \in E : exists(\Delta(e), t)\}$ *and the existence of* $d \in \{+, -, \bullet\}$ *at the time* $t \in \{\texttt{before}, \texttt{after}\}$ *is defined as*

$$exists(d, t) := (t = \texttt{before} \wedge d \neq +) \vee (t = \texttt{after} \wedge d \neq -). \tag{3.2}$$

In later discussions we will need the notion of equivalent variation diffs. In particular, we will call variation diffs which describe the same edit to a variation tree as semantically equivalent. In particular, the projections of semantically equivalent variation diffs need to be the same.

Definition 3.4 (Semantically equivalent variation diffs)**.** *Two variation diffs* D_1 *and* D_2 *are called semantically equivalent iff* $project(D_1, t) \cong project(D_2, t)$ *for all times* $t \in \{\texttt{before}, \texttt{after}\}$*.*

For constructing variation diffs, we need to identify which nodes are unchanged by an edit. This information can be represented by a matching between the variation tree before and after the edit. A matching is a set of pairs of the nodes of each variation tree respectively. An example matching can be seen in Figure 3.2 as red dashed lines. The pairs in this example are $\{(\texttt{root}, \texttt{root}), (\texttt{\#ifdef B}, \texttt{\#ifdef B}), (\texttt{Line}, \texttt{Line})\}$. Note that, in general the matched nodes might have different identifiers. Computing a matching is the first step of constructing variation diffs from variation trees. Most tree diffing algorithms also construct such a matching in their first step, thus we can reuse their existing matching algorithms.

Definition 3.5 (Matching)**.** *A matching* $M = (T_1, T_2, V_1', V_2', m)$ *between two variation trees* $T_1 = (V_1, E_1, r_1, \tau_1, l_1)$ *and* $T_2 = (V_2, E_2, r_2, \tau_2, l_2)$ *consists of a bijective function* $m : V_1' \to V_2'$*, called matching function, with the following properties:*

$$V_1' \subseteq V_1, \quad V_2' \subseteq V_2 \tag{3.3}$$

$$r_1 \in V_1', \quad r_2 \in V_2' \tag{3.4}$$

$$r_2 = m(r_1) \tag{3.5}$$

$$\forall n \in V_1' : \tau_1(n) = \tau_2(m(n)) \wedge l_1(n) = l_2(m(n)) \tag{3.6}$$

We call a matching with $V_1' = \{r_1\}$ *and* $V_2' = \{r_2\}$ *an empty matching.*

Note that matched nodes must not only have the same type but also the same label. This restriction is necessary for variation diffs because an unchanged node has the same label before and after an edit. In contrast, many tree diffing algorithms are not as restrictive and include node label updates in their matchings.

The identifiers of nodes are arbitrary and their concrete values are irrelevant. Just the structure between these identifiers is relevant for our discussion. Hence, we introduce the notion of relabelling a variation tree. Such a relabelling function induces an isomorphism onto variation trees and consequently onto matchings.

Definition 3.6 (Isomorphism between variation trees)**.** *Let $T = (V, E, r, \tau, l, O)$ be a variation tree. A bijective function $relabel : V \to V'$ is called a relabeling function and we say it induces an isomorphism between T and the variation tree*

$$\begin{aligned} T' = (&relabel(V),\\ &\{(relabel(v_1), relabel(v_2)) : (v_1, v_2) \in E\},\\ &relabel(r),\\ &\tau \circ relabel^{-1},\\ &l \circ relabel^{-1}\\ &O \circ relabel^{-1}). \end{aligned}$$

If the relabeling function is irrelevant, we write $T \cong T'$.

Definition 3.7 (Isomorphism between matchings)**.** *Let $M = (T_1, T_2, V_1', V_2', m)$ be a matching between the variation tree $T_1 = (V_1, E_1, r_1, \tau_1, l_1, O_1)$ and the variation tree $T_2 = (V_2, E_2, r_2, \tau_2, l_2, O_2)$. Let $(relabel_1, relabel_2)$ be a pair of two relabeling functions $relabel_1 : V_1 \to V_3$ and $relabel_2 : V_2 \to V_4$ which induce isomorphisms between the variation trees T_1 and T_3 and between the variation trees T_2 and T_4 respectively. We say $(relabel_1, relabel_2)$ induces an isomorphism between M and the matching $M' = (T_3, T_4, relabel(V_1'), relabel(V_2'), relabel_2 \circ m \circ relabel_1^{-1})$. If the relabeling functions are irrelevant, we write $M \cong M'$.*

3.2 Variation Diff Construction Algorithm

Given two variation trees and a matching, we construct a variation diff that represents the edit between the given variation trees. We perform this construction in two phases: First we relabel one of the variation trees to simplify the matching function. Then we union the two variation trees to obtain a variation diff.

We start by defining a relabeling function $relabel_M$ such that $(id, relabel_M)$ induces an isomorphism between a matching M and a simplified matching. Most importantly, the variation trees of the simplified matching will have exactly the matched nodes in common. As consequence, the matching function is the identify function id.

Definition 3.8. *Let $M = (T_1, T_2, V_1', V_2', m)$ be a matching between the variation trees $T_1 = (V_1, E_1, r_1, \tau_1, l_1, O_1)$ and $T_2 = (V_2, E_2, r_2, \tau_2, l_2, O_2)$. We define the function*

$$relabel_M(v) := \begin{cases} m^{-1}(v), & \text{if } v \in V_2' \\ relabel'(v), & \text{if } v \in V_2 \setminus V_2' \end{cases} \tag{3.7}$$

where relabel′ is a bijection which assigns arbitrary but unique labels which are not contained in V_1. For simplicity in the proof of Theorem 3.3, we assume that relabel′ = id if $(V_2 \setminus V_2') \cap V_1 = \emptyset$. Note that some function relabel′ has to exist as we only deal with finite sets.

Lemma 3.1. *Let $T_1 = (V_1, E_1, r_1, \tau_1, l_1, O_1)$ and T_2 be variation trees. Let $M = (T_1, T_2, V_1', V_2', m)$ be a matching. The pair of relabeling functions $(id, relabel_M)$ induces an isomorphism between M and*

$$M' = (T_1, T_3, V_1', V_1', id)$$

with $T_3 = (V_3, E_3, r_3, \tau_3, l_3, O_3)$ and $V_1 \cap V_3 = V_1'$.

Proof. Let $T_1 = (V_1, E_1, r_1, \tau_1, l_1, O_1)$ and $T_2 = (V_2, E_2, r_2, \tau_2, l_2, O_2)$ be variation trees. Let $M = (T_1, T_2, V_1', V_2', m)$ be a matching. The identity function id is bijective and thus a relabeling function. The function $relabel_M : V_2 \to V_3$ is also bijective and thus a relabeling function because

$$relabel_M^{-1}(v) := \begin{cases} m(v), & \text{if } v \in V_1' \\ relabel'^{-1}, & \text{if } v \in V_3 \setminus V_1' \end{cases}$$

is the inverse of $relabel_M$. Note that $V_1' \subseteq V_1$ follows from Definition 3.5 and thus per Definition 3.8 $relabel' : V_2 \setminus V_2' \to V_3 \setminus V_1'$. Hence id and $relabel_M$ induce isomorphisms between T_1 and T_1 and between T_2 and $T_3 = (V_3, E_3, r_3, \tau_3, l_3, O_3)$ with $V_3 = relabel_M(V_2)$ and thus $V_1 \cap V_3 = V_1 \cap relabel_M(V_2) = V_1'$ because we defined $relabel_M(V_2') = V_1'$ and $relabel_M(V_2 \setminus V_2') \cap (V_1 \cup V_2) = \emptyset$. In summary, $(id, relabel_M)$ induces an isomorphism between M and

$$\begin{aligned} M' &= (T_1, T_3, V_1', relabel_M(V_2'), relabel_M \circ m) \\ &= (T_1, T_3, V_1', V_1', id). \end{aligned}$$

□

After relabelling, we can union the relabeled the variation trees and mark all nodes which are matched as unchanged. All nodes and edges which are not present in both variation trees are marked as deleted or inserted respectively.

Definition 3.9 (Construction)**.** *Let $T_1 = (V_1, E_1, r_1, \tau_1, l_1, O_1)$ and T_2 be variation trees. Let $M = (T_1, T_2, V_1', V_2', m)$ be a matching. We define our variation diff construction algorithm as $construct(M) := (V, E, r, \tau, l, \Delta, O_{before}, O_{after})$ where*

$$V := V_1 \cup V_3 \tag{3.8}$$

$$E := E_1 \cup E_3 \tag{3.9}$$

$$r := r_1 \tag{3.10}$$

$$\tau(n) := \begin{cases} \tau_1(n), & \text{if } n \in V_1 \\ \tau_3(n), & \text{if } n \in V_3 \setminus V_1 \end{cases} \tag{3.11}$$

$$l(n) := \begin{cases} l_1(n), & \text{if } n \in V_1 \\ l_3(n), & \text{if } n \in V_3 \setminus V_1 \end{cases} \tag{3.12}$$

$$\Delta(x) := \begin{cases} \bullet, & x \in V_1 \cap V_3 \vee \text{if } x \in E_1 \cap E_3 \\ +, & x \in V_3 \setminus V_1 \vee \text{if } x \in E_3 \setminus E_1 \\ -, & x \in V_1 \setminus V_3 \vee \text{if } x \in E_1 \setminus E_3 \end{cases} \tag{3.13}$$

$$O_{before} := O_1 \tag{3.14}$$

$$O_{after} := O_3 \tag{3.15}$$

and where $(id, relabel_M)$ *induces an isomorphism between the matching* M *and the matching* $M' = (T_1, T_3, V_1', V_1', id)$ *with* $T_3 = (V_3, E_3, r_3, \tau_3, l_3, O_3)$.

To prove that *construct* is consistent, we have to prove that $D = construct(M)$ is indeed a variation diff, i.e. that $project(D, t)$ is a variation tree for all $t \in \{\texttt{before}, \texttt{after}\}$. Furthermore, we want to prove that D is consistent with the intended semantic of actually representing an edit between the given variation trees, i.e. we want to have $project(construct(M), \texttt{before}) \cong T_1$ and $project(construct(M), \texttt{after}) \cong T_2$ for any matching $M = (T_1, T_2, V_1', V_2', m)$. As special case of our choice of $relabel_M$ and because we only relabel the variation after the edit, we actually get the stronger guarantee $project(D, \texttt{before}) = T_1$ for the variation tree before the edit. For later reuse, we explicitly state the relabel functions used to induce the isomorphism.

Theorem 3.1 (Consistency)**.** *Given a matching* M *between two variation trees* T_1 *and* T_2*,* $D = construct(M)$ *is a variation diff where* $project(D, \texttt{before}) = T_1$ *and* $project(D, \texttt{after}) = T_3 \cong T_2$ *are variation trees. The isomorphism between* M *and the matching* $M' = (T_1, T_3, V_1', V_1', id)$ *is induced by* $(id, relabel_M)$.

Proof. Let $M = (T_1, T_2, V_1', V_2', m)$ be a matching between the two variation trees $T_1 = (V_1, E_1, r_1, \tau_1, l_1, O_1)$ and $T_2 = (V_2, E_2, r_2, \tau_2, l_2, O_2)$. For notation, we define $D := (V, E, r, \tau, l, \Delta, O_{before}, O_{after}) := construct(M)$. According to Lemma 3.1, the relabeling functions $(id, relabel_M)$ induce an isomorphism between M and the matching $M' = (T_1, T_3, V_1', V_1', id)$ with $T_3 = (V_3, E_3, r_3, \tau_3, l_3, O_3)$ and $T_2 \cong T_3$. Hence, we have

$$\begin{aligned} project(D, \texttt{after}) &\overset{(3.1)}{=} (V', E', r, \tau|_{V'}, l|_{V'}, O_{\texttt{after}}) \\ &\overset{(3.10)}{=} (V', E', r_1, \tau|_{V'}, l|_{V'}, O_{\texttt{after}}) \\ &\overset{(3.5)}{=} (V', E', r_3, \tau|_{V'}, l|_{V'}, O_{\texttt{after}}) \\ &\overset{(3.11)}{=} (V', E', r_3, \tau, l|_{V'}, O_{\texttt{after}}) \\ &\overset{(3.12)}{=} (V', E', r_3, \tau, l, O_{\texttt{after}}) \\ &\overset{(3.15)}{=} (V', E', r_3, \tau, l, O_3) \\ &= (V_3, E_3, r, \tau, l, O_3) \\ &= T_3 \\ &\cong T_2 \end{aligned}$$

because

$$
\begin{aligned}
V' &= \{v \in V : exists(\Delta(v), \texttt{after})\} \\
&\overset{(3.2)}{=} \{v \in V : \Delta(v) \neq -\} \\
&\overset{(3.8)}{=} \{v \in V_1 \cup V_2 : \Delta(v) \neq -\} \\
&= \{v \in V_1 : \Delta(v) \neq -\} \\
&\quad \cup \{v \in V_2 : \Delta(v) \neq -\} \\
&\overset{(3.13)}{=} (V_1 \cap V_3) \cup (V_3 \setminus V_1) \\
&= V_3
\end{aligned}
\qquad
\begin{aligned}
E' &= \{e \in E : exists(\Delta(e), \texttt{after})\} \\
&\overset{(3.2)}{=} \{e \in E : \Delta(e) \neq -\} \\
&\overset{(3.9)}{=} \{e \in E_1 \cup E_2 : \Delta(e) \neq -\} \\
&= \{e \in E_1 : \Delta(e) \neq -\} \\
&\quad \cup \{e \in E_2 : \Delta(e) \neq -\} \\
&\overset{(3.13)}{=} (E_1 \cap E_3) \cup (E_3 \setminus E_1) \\
&= E_3
\end{aligned}
$$

Analogous, we can prove $project(D, \texttt{before}) = T_1$. As a result of the projections being variation trees, we proved that $construct(D)$ is a variation diff. □

3.3 Inverse of our Construction Algorithm

It is possible to reconstruct the matching, which was used during construction, from a variation diff. Consider Figure 3.2 as an example. The matched nodes are exactly the nodes which are unchanged. Indeed, the matching obtained by *matching* is equivalent to the matching we use in the construction of variation diffs. Hence, we can proof that *construct* is injective.

Definition 3.10 (Reconstruct Matching)**.** *Let $D = (V, E, r, \tau, l, \Delta, O_{before}, O_{after})$ be a variation diff. We define*

$$matching(D) := (project(D, \texttt{before}), project(D, \texttt{after}), V', V', id) \tag{3.16}$$

with $V' = \{n \in V : \Delta(n) = \bullet\}$.

Theorem 3.2 (Injectivity)**.** *Given a matching M between two variation trees T_1, T_2 with $D = construct(M)$ we have*

$$matching(construct(M)) = M'$$

with a matching $M' \cong M$ between two variation trees $T_1 \cong project(D, \texttt{before})$ and $T_2 \cong project(D, \texttt{after})$.

Proof. Let $M = (T_1, T_2, V_1', V_2', m)$ be a matching between the two variation trees $T_1 = (V_1, E_1, r_1, \tau_1, l_1, O_1)$ and T_2. We denote the constructed variation diff by $D = (V, E, r, \tau, l, \Delta, O_{before}, O_{after}) = construct(M)$. According to Lemma 3.1, the relabeling functions $(id, relabel_M)$ induce an isomorphism between M and the matching $M' = (T_1, T_3, V_1', V_1', id)$ with $T_3 = (V_3, E_3, r_3, \tau_3, l_3, O_3)$ and $V_1 \cap V_3 = V_1'$.

Expanding Definition 3.10 results in

$$
\begin{aligned}
matching(D) &= (project(D, \texttt{before}), project(D, \texttt{after}), V', V', id) \\
&\overset{\text{Theorem 3.1}}{=} (T_1, T_3, V', V', id) \\
&= (T_1, T_3, V_1', V_1', id) \\
&= M'
\end{aligned}
$$

because

$$\begin{aligned} V' &= \{n \in V : \Delta(n) = \bullet\} \\ &\overset{(3.8)}{=} \{n \in V_1 \cup V_3 : \Delta(n) = \bullet\} \\ &\overset{(3.13)}{=} \{n \in V_1 \cup V_3 : n \in V_1 \cap V_2 \vee n \in E_1 \cap E_2\} \\ &= \{n \in V_1 \cup V_3 : n \in V_1 \cap V_2\} \\ &= V_1 \cap V_3 \\ &= V_1'. \end{aligned}$$

□

Moreover, our construction algorithm *construct* is not only injective, but also surjective and thus bijective. Hence, the inverse of *construct* is *matching*.

Theorem 3.3 (Surjectivity)**.** *Let D be a variation diff. It holds that*

$$D = construct(matching(D)).$$

Proof. Let $D = (V, E, r, \tau, l, \Delta, O_{\texttt{before}}, O_{\texttt{after}})$ be a variation diff.

First, we expand Definition 3.10 and Definition 3.3 to get

$$\begin{aligned} matching(D) &\overset{(3.16)}{=} (project(D, \texttt{before}), project(D, \texttt{after}), V', V', id) \\ &\overset{(3.1)}{=} ((V_1, E_1, r, \tau|_{V_1}, l|_{V_1}, O_{\texttt{before}}), (V_2, E_2, r, \tau|_{V_2}, l|_{V_2}, O_{\texttt{after}}), V', V', id) \\ &=: (T_1, T_2, V', V', id) \\ &=: M \end{aligned}$$

with

$$\begin{aligned} V_1 &= \{v \in V : exists(\Delta(v), \texttt{before})\} \overset{(3.2)}{=} \{v \in V : \Delta(v) \neq +\} \\ V_2 &= \{v \in V : exists(\Delta(v), \texttt{after})\} \overset{(3.2)}{=} \{v \in V : \Delta(v) \neq -\} \\ E_1 &= \{e \in E : exists(\Delta(e), \texttt{before})\} \overset{(3.2)}{=} \{e \in E : \Delta(e) \neq +\} \\ E_2 &= \{e \in E : exists(\Delta(e), \texttt{after})\} \overset{(3.2)}{=} \{e \in E : \Delta(e) \neq -\} \\ V' &= \{n \in V : \Delta(n) = \bullet\} \end{aligned}$$

Next, we have to relabel M with $(id, relabel_M)$ to finally apply Definition 3.9. We know

$$\begin{aligned} (V_2 \setminus V') \cap V_1 &= (\{v \in V : \Delta(v) \neq -\} \setminus \{n \in V : \Delta(n) = \bullet\}) \cap \{v \in V : \Delta(v) \neq +\} \\ &= \{v \in V : \Delta(v) = +\} \cap \{v \in V : \Delta(v) \neq +\} \\ &= \emptyset \end{aligned}$$

because $\Delta(v) \in \{+, -, \bullet\}$. Hence, we get $relabel_M = id$ because in Definition 3.8 we define $relabel' = id$ for this special case to simplify this proof. Hence, relabeling M with $(id, relabel_M) = (id, id)$ does not change anything. Nonetheless, we introduce $T_3 := (V_3, E_3, r, \tau, l, O_{\texttt{after}}) := T_2$ for consistent naming with Definition 3.9.

Finally we can fully expand Definition 3.9 to get

$$construct(matching(D)) = (\tilde{V}, \tilde{E}, \tilde{r}, \tilde{\tau}, \tilde{l}, \tilde{\Delta}, \tilde{O}_{\texttt{before}}, \tilde{O}_{\texttt{after}})$$

with

$$\tilde{V} \overset{(3.8)}{=} V_1 \cup V_3 = V$$

$$\tilde{E} \overset{(3.9)}{=} E_1 \cup E_3 = E$$

$$\tilde{r} \overset{(3.10)}{=} r$$

$$\tilde{l}(n) \overset{(3.12)}{=} \begin{cases} l|_{V_1}(n), & \text{if } n \in V_1 \\ l|_{V_2}(n), & \text{if } n \in V_2 \setminus V_1 \end{cases} = l(n)$$

$$\tilde{\tau}(n) \overset{(3.11)}{=} \begin{cases} \tau|_{V_1}(n), & \text{if } n \in V_1 \\ \tau|_{V_2}(n), & \text{if } n \in V_2 \setminus V_1 \end{cases} = \tau(n)$$

$$\tilde{\Delta}(x) \overset{(3.13)}{=} \begin{cases} \bullet, & \text{if } x \in V_1 \cap V_2 \vee x \in E_1 \cap E_2 \\ +, & \text{if } x \in V_2 \setminus V_1 \vee x \in E_2 \setminus E_1 \\ -, & \text{if } x \in V_1 \setminus V_2 \vee x \in E_1 \setminus E_2 \end{cases} = \Delta(x)$$

$$\tilde{O}_{\texttt{before}} \overset{(3.14)}{=} O_{\texttt{before}}$$

$$\tilde{O}_{\texttt{after}} \overset{(3.15)}{=} O_{\texttt{after}}$$

In summary we have

$$\begin{aligned} construct(matching(D)) &= (\tilde{V}, \tilde{E}, \tilde{r}, \tilde{\tau}, \tilde{l}, \tilde{\Delta}, \tilde{O}_{\texttt{before}}, \tilde{O}_{\texttt{after}}) \\ &= (V, E, r, \tau, l, \Delta, O_{\texttt{before}}, O_{\texttt{after}}) \\ &= D \end{aligned}$$

□

Corollary. *Variation diffs are a different representation for matchings between variation trees.*

An important implication of this corollary is that there is no quality optimisation potential in the variation diff construction alone. The quality of a variation diff depends solely on the matching algorithm. There might be potential optimisations if variation diffs are extended to encode which nodes are moved using edit scripts because such edit scripts are not unique. However, there are already algorithms for minimizing some metrics related to edit scripts (e.g. edit cost or edit script size) [9]. In contrast, computing an optimal matching is NP-hard when optimizing edit scripts including node movements [4]. Thus, matchings can only be approximated in practice.

An additional implication of this corollary is that the variation diffs constructed by the state-of-the-art algorithm developed by Viegener [44] also encode a matching. This matching is implicitly provided by the line-based diffing algorithm. Hence, the algorithm by Viegener [44] is a special case of the *construct* function using a specific matching algorithm and a line-based diff as intermediate representation of the variation trees and the matching. The set of matched lines is simply the set of nodes without modifications

in the variation diff. Furthermore, both variation diff construction algorithms have linear time and space requirements when excluding the match generation (the line differ in the case of Viegeners Algorithm). Hence, the main benefits of our variation diff construction algorithms are its ability to use any matching algorithm during construction and its ability to handle nested artifact nodes such as ASTs.

3.4 Summary

In this chapter, we have extended the definition of variation trees and variation diffs with child ordering to correctly identity node movements with the same parent before and after an edit. We introduced the notion of semantically equivalent variation diffs for use in Chapter 4. We formally defined matchings between variation diffs and isomorphisms between them. Finally, we introduced our algorithm for construction variation diffs from matchings between variation trees. Formal analysis of our variation diff construction algorithm revealed that our algorithm is able to construct arbitrary variation diffs using arbitrary matching algorithms, taking advantage of the full flexibility variation diffs provide.

4. Quality Dimensions and Metrics for Variation Diffs

In this chapter, we discuss how variation diffs can differ and identify four quality dimensions. We present multiple metrics for comparing semantically equivalent variation diffs qualitatively. It is possible to reuse the quality metrics of line and tree differs but we also define metrics directly on variation diffs.

We start in Section 4.1 by discussing how variation diffs can differ. Then, we screen comparison metrics for comparing tree differs which can be used to evaluate the underlying matching of variation diffs in Section 4.2.1. In Section 4.2.2, we introduce two simple and one practical metric on variation diffs themselves.

4.1 Variation Diff Quality Dimensions

We identified four dimensions which define the quality of a variation diff: The object language, metalanguage, granularity, and diff language. We adopt the terms object language and metalanguage from Erwig et al. [15] and Walkingshaw [45] who introduce this terminology to differentiate between their choice calculus (metalanguage) and the underlying artifacts (object language). Although all four quality dimensions influence variation diffs, a variation diff construction algorithm is only able to influence the diff language according to our definition of semantically equivalent variation diffs. However, consider, for example, that the granularity can be freely chosen when constructing variation trees. Thus, alternative definitions of equivalent variation diffs could allow differences in quality dimensions other than the diff language.

"An object language is a language used to describe a single, non-variational artifact." [45] A typical object language in software product lines is the C language without preprocessor conditionals. Variation trees and thus variation diffs include the artifacts of a software product lines in the form of artifact nodes. However, the edits that variation diffs describe can be influenced by the structure of the object language. For example, the source code in Figure 4.1a has duplicate code which could be extracted into a function as seen in Figure 4.1b. Edits to the duplicate code cause more differences in the variation

trees which reflect into the variation diffs. In our example, two behaviorally equivalent edits of adding a tab character before the printed byte can be seen in Figure 4.1c and Figure 4.1d. The variation diff in Figure 4.1d contains less changed nodes as there was less duplication in the object language. In contrast, the variation diff in Figure 4.1c has more changed nodes and thus represents more edits as each duplicated artifact is edited. In general, the artifacts of a software product line are fixed when constructing variation diffs because the resulting variation diff would describe a different edit if the artifacts were changed. Hence, differences in the object language are not suitable for comparing the quality of different variation diffs.

"A metalanguage (also called a variation language), is a language used to describe the variability in a variational artifact." [45] In the case of variation trees and variation diffs the metalanguage consists of the annotation nodes and their structure. An example quality of a metalanguage is the sharing of commonalities between different variants [15]. In contrast to the choice calculus of Erwig et al. [15] variation trees and thus variation diffs do not have an explicit sharing operator. Nonetheless, a limited amount of sharing can be achieved as Figure 4.2 shows. In Figure 4.2a, the artifacts `C` and `D` and their annotations are duplicated for the variants including the feature `A` and the variations including the feature `B` but excluding `A`. Sharing can be achieved, for example by extracting the duplicated subtree and adding missing annotations as in Figure 4.2b. Both variation trees represent a software product line with equal variants. However if a duplicated node is edited, the variation diff required more edited nodes. Thus, the benefit of Figure 4.2b in comparison to Figure 4.2a is similar to the benefit in the object language example in Figure 4.1. In cases where the metalanguage is explicitly annotated in the source, for example in C based software product lines, it is generally not desired to construct variation trees and variation diffs which represent different annotations. However, the quality dimension provided by the metalanguage may be flexible or subject to automatic refactoring in some applications.

Given more detailed input data, we can include more details in a variation tree or variation diff. The granularity of a variation tree and variation diff describes how detailed the graph structure is. The granularity can range from, for example, one artifact node per file, over one artifact node per line, to full abstract syntax trees. Note, however, that variation trees and variation diffs do not have to have a consistent granularity over the whole graph. Although variation diffs with different granularity cannot be semantically equivalent, they may still represent the same edit. For example, in Figure 4.3 we depict two variation trees which describe the same edit. On the one hand, Figure 4.3a describes the edit using line granularity. On the other hand, we represent the same edit as a variation diff with the granularity of a simplified abstract syntax tree in Figure 4.3b. In contrast to Viegener's algorithm [44], our construction algorithm can handle arbitrary variation tree granularities. Thus, one can choose a granularity which suits the needs of a particular application.

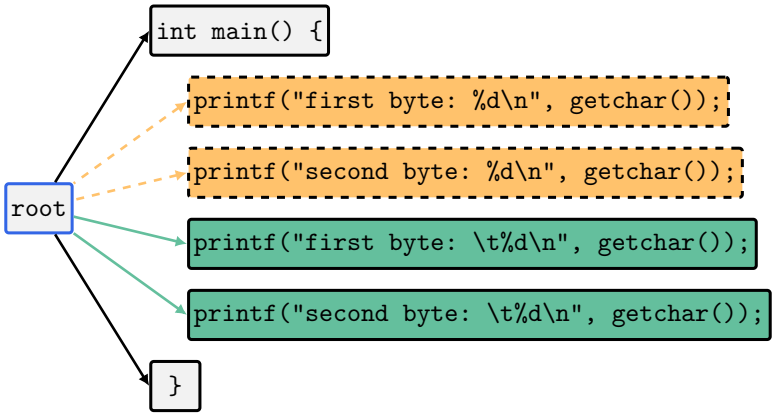
Finally, a diff language interprets two variation trees as an edit. In particular, it specifies which nodes are unchanged, moved, deleted, or removed. The diff language of variation diffs is the time of existence of the nodes and edges (i.e. the colors in our examples). Of the discussed quality dimensions, the diff language is the only quality dimension not present in variation trees. Moreover, semantically equivalent variation diffs differ only in the diff language. Hence, the diff language corresponds exactly to the matching used

```c
int main() {
        printf("first byte: %d\n", getchar());
        printf("second byte: %d\n", getchar());
}
```

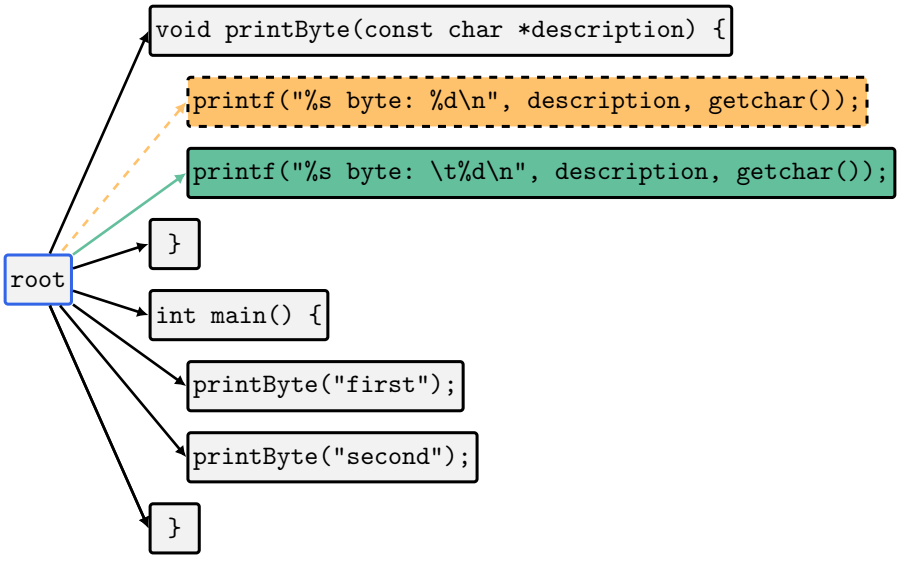
(a) Example of duplicate code in C

```c
void printByte(const char *description) {
        printf("%s byte: %d\n", description, getchar());
}
int main() {
        printByte("first");
        printByte("second");
}
```

(b) Example for reducing the duplicate code of Figure 4.1a

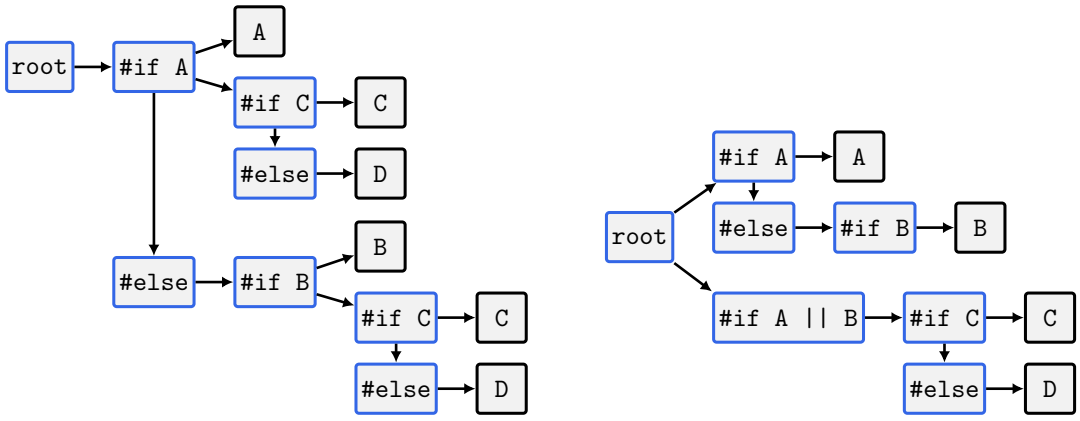


(c) Variation diff of an edit to Figure 4.1a



(d) Variation diff of an edit to Figure 4.1b

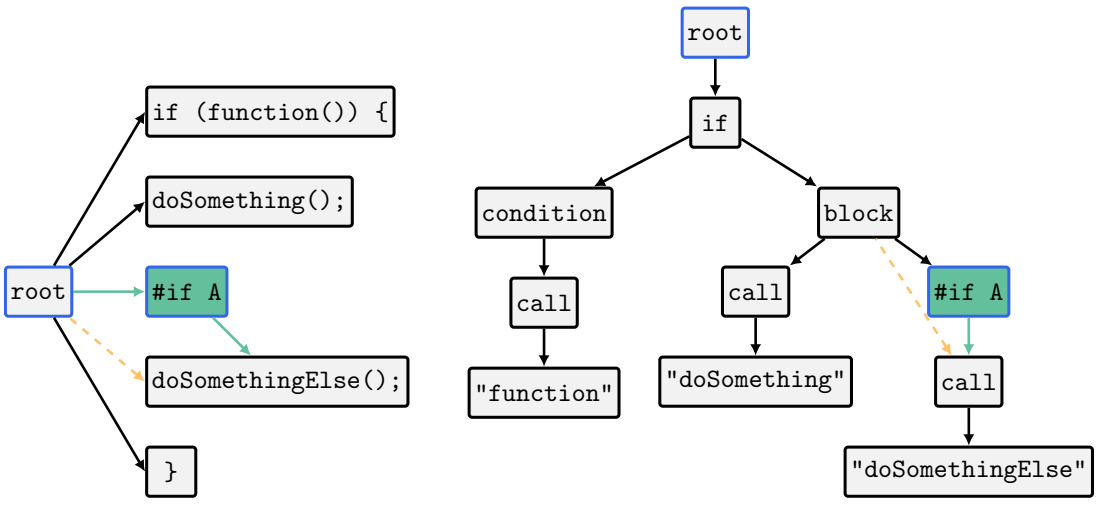
Figure 4.1: Example C code and variation diffs with and without duplication in the object language



(a) Variation tree with duplication in the metalanguage

(b) Variation tree with sharing in the metalanguage

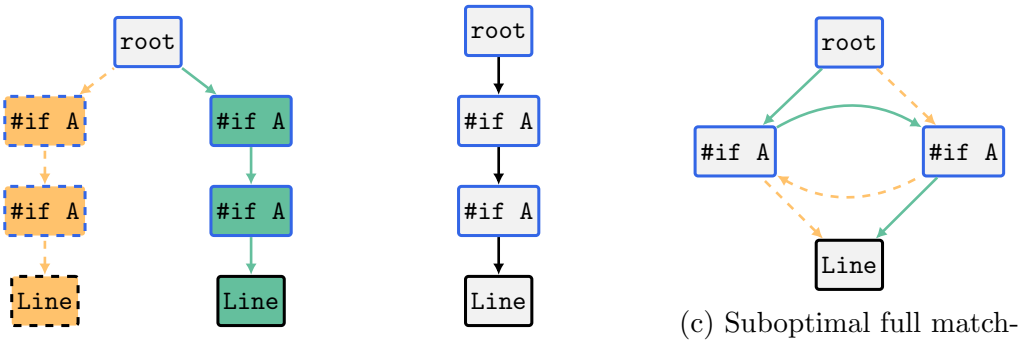
Figure 4.2: Example of sharing in the metalanguage of variation trees



(a) Line granularity

(b) AST granularity

Figure 4.3: Example variation diffs with different granularity describing the same edit



(a) Empty matching (b) Optimal matching (c) Suboptimal full matching

Figure 4.4: Example variation diffs differing only in the diff language (i.e. their matching)

during the construction of a variation diff because of the bijection between variation diffs and matchings between variation trees. For example, the variation diff in Figure 4.4a corresponds to the empty matching, whereas the variation diff in Figure 4.4b corresponds to a matching of maximal size. Although these variation diffs are semantically equivalent, their interpretation is "everything is deleted and new artifacts are reinserted", in the case of Figure 4.4a and "nothing changed" in the case of Figure 4.4b. When a node is unchanged instead of deleted and inserted, we call that node shared. As shown with this example, diff language sharing can be used as a quality indicator for semantically equivalent variation diffs.

The amount of sharing in the diff language is not the only metric one should optimize for. As an example when this might lead to bad quality, compare the variation diff in Figure 4.4c to the variation diff in Figure 4.4b. Both are constructed using a maximal matching but Figure 4.4b one doesn't represent any changes whereas Figure 4.4c represents the exchange of the duplicate annotation `#if A`. Such unfortunate matching possibilities get more numerous with duplication in the object and metalanguage because they require duplicate nodes. Moreover, there are examples where diffs get harder to understand when the diff language sharing is increased [12]. For example, consider Figure 4.5a, which shows a line diff for simplicity. The empty lines of the deleted code are matched with the empty lines of the new code, making the diff, in our opinion, harder to read in comparison to Figure 4.5b. However, this might be beneficial for some other applications like computing the edit cost. Hence, the quality of variation diffs, or diffs in general, should be compared using application-specific metrics.

4.2 Metrics for the Quality of Variation Diffs

A quality metric is a function comparing two variation diffs. We use metrics to establish the optimisation goal of finding a variation diff which is, according to the chosen metric, better to as many other variation diffs as possible. There are many different quality metrics imaginable and their fitness depends on the concrete application. However, we only discuss metrics comparing semantically equivalent variation diffs.

4.2.1 Metrics from Line and Tree Differs

Line and tree differs can be compared by the size of the line matching they produce [16, 35]. Furthermore, the size of the matching corresponds directly to the amount of sharing

```
-int rollDice() {
+int getRandomNumer() {
+        static bool seeded
+                = false;

-        // chosen by fair
-        // dice roll.
-        int n = 4;
+        if (!seeded) {
+                srand(seed);
+                seeded = true;
+        }

-        // guaranteed to
-        // be random.
-        return n;
+        return rand();
 }
```

(a) Matched whitespace

```
-int rollDice() {
-
-        // chosen by fair
-        // dice roll.
-        int n = 4;
-
-        // guaranteed to
-        // be random.
-        return n;
+int getRandomNumer() {
+        static bool seeded
+                = false;
+
+        if (!seeded) {
+                srand(seed);
+                seeded = true;
+        }
+
+        return rand();
 }
```

(b) Unmatched whitespace

Figure 4.5: Example line diff demonstrating a possible problem with too much sharing

in the diff language. Thus, the simplest way to compare variation diffs is to compare the size of the matching used for their construction. However, maximizing the size of the matching can result in matchings hampering the understandability of a diff as discussed in the previous section.

Line and tree differs do not primarily produce matchings but process them further to create edit scripts [7, 9, 12, 13, 16, 17, 18, 21, 22, 29, 30, 31, 33, 34, 35]. Thus, usually edit scripts, in particular their edit cost, instead of matchings are compared [16, 18, 21, 31]. Edit scripts are a list of edit actions which, when performed on the data structure before an edit, create the data structure after the edit. The edit cost of an edit script is calculated by assigning weights to each edit action depending on their operands and summing these costs. Typical edit actions of line differs are line addition, deletion, and movement [7, 22, 33, 34, 35]. Typical tree differs generalize these operations to nodes and add some more operations resulting in edit actions like node addition, deletion, movement, and update [9, 12, 13, 16, 17, 18, 21, 29, 30, 31]. As the quality of edit scripts depends on the matching it is possible to reuse the same metric for comparing variation diffs through their matchings. However, the diff language of edit scripts and variation diffs might provide different optimization incentives.

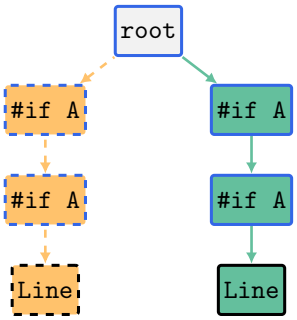
Many researchers use manual comparison as a quality metric [12, 13, 16, 18, 27, 31]. They show that their metric optimizes for edit scripts which are easier to understand than when using edit costs as a quality metric. Another approach is to classify edits with a goal oriented approach [17]. Fluri et al. [17] define optimal edit scripts by hand and evaluate an edit script by comparing it to their optimum. Both approaches can be applied to variation diffs either directly by comparing variation diffs or indirectly by comparing edit scripts generated from the matching of variation diffs. In contrast to all other metrics we discuss, both of these metrics are computed by humans and are not algorithmically computable.

All quality metrics in this section have in common that they have been used for optimising the matching algorithms of line and tree differs. As proven in Chapter 3, the underlying matching is the only variable of variation diffs with the same semantics. If we use these metrics to compare variation diffs, we effectively compare the line and tree diffing algorithms used in our implementation. Hence, it is not meaningful to reevaluate these metrics in the context of variation diffs as the results should be equivalent to the original applications of these metrics. Furthermore, the metrics used for line and tree diffing algorithms might not actually define the same quality one expects from variation diffs as small edit costs and understandability might not be the only use cases for variation diffs.

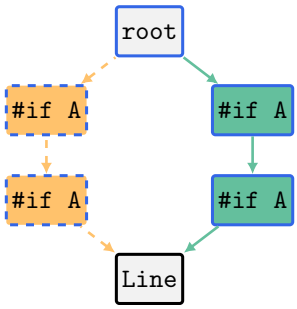
4.2.2 Metrics on Variation Diffs

Comparing the number of nodes in variation diffs is equivalent to comparing the size of the matching used for constructing these variation diffs. Consider a sequence of variation diffs starting with an empty matching successively adding one matching until some arbitrary variation diff. Each time a matching is added, we decrease the number of nodes in the variation diff by one because we join an added and a removed node into an unchanged node. In particular, it holds that

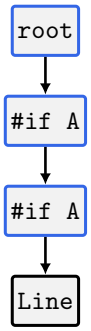
$$|D| = |project(D, \texttt{before})| + |project(D, \texttt{after})| - |matching(M)|$$



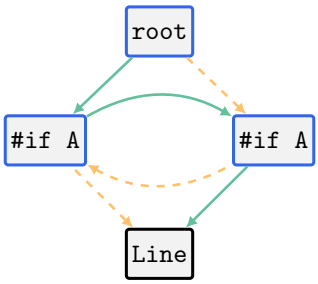
(a) Empty matching, node count: 7, edge count: 6



(b) Improved matching, node count: 6, edge count: 6



(c) Optimal matching, node count: 4, edge count: 3



(d) Suboptimal full matching, node count: 4, edge count: 6

Figure 4.6: Comparison of the node count and edge count metric between four example variation diffs

for a variation diff D where $|\cdot|$ denotes the number of nodes in a graph or the size of a matching. Alternatively, the number of unchanged nodes can be counted, which is exactly equal to the size of the matching. Thus, minimizing the number of nodes is equivalent to maximizing the matching.

The edge count is another metric on variation diffs. Alternatively, as with the case of the node count, it is possible to count the number of unchanged edges. Intuitively, the edge count is more conservative than the node count because an edge between two nodes can only be marked as unchanged if both nodes are unchanged. For example, the variation diff in Figure 4.6a is inferior to the variation diff in Figure 4.6b when compared using the node count but both have the same edge count. However, it is also possible that the node count is the same but the edge count differs. For example, the node count in Figure 4.6c is the same as in Figure 4.6d but the edge count differs. In our experience, optimizing according to the edge count flags less optimization potential but results in less suboptimal changes.

One problem case of the edge count metric is the dependence on the granularity. Consider variation diffs without artifact nesting, for example line granular variation diffs as in Figure 4.2. The influence of a matching between two artifact nodes depends on their parents. Hence, without artifact nesting, matching two artifacts may only influence the edge count if their parent feature annotation is marked as unchanged. Simple changes to presence conditions might result in many unmatched nodes which are not reflected

specificity	edit class
1	RemWithMapping, AddWithMapping
2	RemFromPC, AddFromPC
3	Reconfiguration
4	Specialisation, Generalisation
5	Refactoring
6	Untouched

Table 4.1: Proposed partial order for the edit classed presented by Bittner et al. [6]

in the edge count metric. Thus, the edge count metric is problematic when used on flat variation diffs and variation diffs without artifact nesting.

An application-specific metric might provide a better optimization goal than the general metrics presented so far. As example, we consider the edit classification by Bittner et al. [6] which classifies each artifact in a variation diff into one of nine edit classes. To compute this metric, we classify each artifact in the two variation diffs to be compared. For better results, we do not compare the edit class count directly but compare the edit class flows. Hence, we compare the edit classifications between the artifacts in the projections of the variation diffs and count the number edit classification pairs which we call edit class flow. By projecting both variation diffs we obtain equal variation trees with possibly different artifact classifications. Thus, the edit classification of Bittner et al. [6] can be used to compare variation diffs, for example using Sankey-diagrams.

For using the edit classification of Bittner et al. [6] as a metric, we need to assess which edit class flows improve the quality of a variation diff. We propose the partial order of Bittner et al. [6] edit classes as seen in Table 4.1 which discriminates different edit class flows. Intuitively, we consider an edit class E_1 more specific as an edit class E_2 if the probability that an artifact matched by E_1 can be changed to a match of E_2 by modifying the diff language is higher than the probability that a match of E_2 can be changed to a match of E_1. For example, an artifact classified as RemFromPC, which has an annotation, which is not the root, as parent, can be classified as RemWithMapping by marking the annotation above the artifact node as deleted (and adding an equivalent inserted node). The contrary is only possible if there exists an inserted annotation which can be matched with the annotation above the artifact node. Hence, we consider RemFromPC more specific than RemWithMapping. We did not formally prove these probability differences but are confident that our proposed ordering serves our intention well.

Analogous to the edit weights used to calculate the cost of an edit script, we can assign values to each edit class flow. For a quality comparison, we can differentiate between positive, neutral, and negative edit flows. The net amount of flow between different edit classes represents the quality difference between the compared variation diff. If an edit was classified with a more specific edit class as before, this flow is considered positive. Negative flows are defined respectively and if both classifications have equal specificity, it is considered neutral. Hence, we can compare variation diffs qualitatively by comparing the amount if positive and negative edit class flows between them.

4.3 Summary

In this chapter, we presented the four quality dimensions object language, metalanguage, granularity, and diff language for variation diffs. We explained our focus on the diff language and introduced the notion of sharing in the diff language. Furthermore, we discussed how metrics used for line and tree differs can be repurposed to compare variation diffs. Finally, we presented an application-specific quality metric for variation diffs in the domain of classifying edits to software product lines.

5. Experimental Evaluation

In this chapter, we validate and evaluate the output and performance of our variation diff construction algorithm to assess its practicality and quality. Specifically, the goals of our evaluation are as follows:

RG1 Verify the consistency and correctness of our construction algorithm.

RG2 Evaluate the practicality and scalability of our construction algorithm.

RG3 Compare a line, tree, and hybrid matching algorithm regarding the quality of the variation diffs constructed from their matchings.

We can confirm our consistency proof from Chapter 3 (RG1) by verifying the consistency of all variation diffs that are constructed by our construction algorithm, Furthermore, we verify the correctness of our construction algorithm (RG1) by checking, for each edit, that different matchings result in semantically equivalent variation diffs. Using the metrics presented in Chapter 4, we can compare a line, tree and hybrid matching algorithm regarding the quality of the variation diffs constructed using their matchings. This lets us evaluate the fitness of these matching algorithms for constructing variation diff with line granularity (RG3). In addition, we benchmark the different matching and construction algorithms to evaluate how practical and scalable our variation diff construction algorithm is (RG2).

First of all, we explain our experiment setup in Section 5.1. We present the results of our experiment in Section 5.3. Finally, in Section 5.4, we interpret the results and discuss threats to the validity of our results.

5.1 Experiment Setup

We automatically extract the Git history of three open-source software product lines. Then we construct three variation diffs using different algorithms and accumulate the construction runtime and some quality metrics of the constructed variation diffs. We ran

our evaluation on an Intel©Xeon©CPU with 2.40GHz and 32 threads using GNU/Linux as operating system. Specifically, we used Ubuntu 22.04.1 LTS (Jammy Jellyfish) and OpenJDK 17.0.6. For reducing the evaluation time, we analyze multiple commits in parallel and do not use this machine for other tasks during the evaluation.

We check invariants of all variation diffs and intermediate matchings to verify their consistency (RG1). Checked invariants include the graph structure, time annotations for nodes and edges, valid annotation nesting, and that matchings are a set of matched nodes. The correctness of our construction algorithm (RG1) is checked by comparing both projections of each variation diff with each other. Any differences would be due to semantic inequalities.

In contrast to our construction algorithm, Viegener's algorithm is only capable of parsing variation diffs with line granularity from the line diff of a single file. Hence, we apply all variation diff construction algorithms to each patch (all edits to a single file) of a commit separately. To enable qualitative comparisons, we also limit our construction algorithm to line granularity although it could parse the whole project with AST granularity. Note that this implies that our whole experiment is specific to variation diffs with line granularity.

For each processed patch, we construct three variation diffs. The first variation diff is constructed by Viegener's algorithm [44], which is the first and currently only variation diff construction algorithm, Viegener's algorithm constructs variation diffs using the implicit matching provided by the Git line differ. For the second and third variation diff, we us our construction algorithm with two different matching algorithms. The tree matcher GumTree [16] provides the matching for the second variation diff. The third variation diff is constructed using a hybrid matcher as introduced by Matsumoto et al. [31] which combines a line and tree matcher. The construction of all three variation diffs is benchmarked to answer RG2 and the results are compared using the metrics presented in Chapter 4 to answer RG3.

There are many alternative tree matching algorithms [9, 12, 13, 17, 18, 21, 38]. Our selection of tree matchers is based on the availability of many different matching algorithms in the GumTree library [1]. Unfortunately, some matching algorithms in the GumTree library are buggy (XY [10], MTDIFF optimisations [13]) or require too much memory (RTED [38]). Hence, we only use the original GumTree matcher.

The hybrid algorithm of Matsumoto et al. [31] is a notable compromise because there is an alternative hybrid matching algorithm [51] with very promising performance. We use the former matching algorithm because there is no implementation of these algorithms in GumTree and we can use the bijection between variation diffs and tree matchings for its implementation. The original hybrid matcher by Matsumoto et al. [31] consists of three phases. First, it generates a line matching between two files. Then, it parses these files into fine-granular trees and transfers the line matching onto the trees by matching nodes which are contained in a matched line. Finally, it runs GumTree on the transferred matching which is able extend the matching (e.g. detect moved nodes). In contrast, we want trees with line granularity, so we can directly use the matching of a line differ as input for GumTree. In practice, we implemented the hybrid matching algorithm as an

[1] https://github.com/GumTreeDiff/gumtree

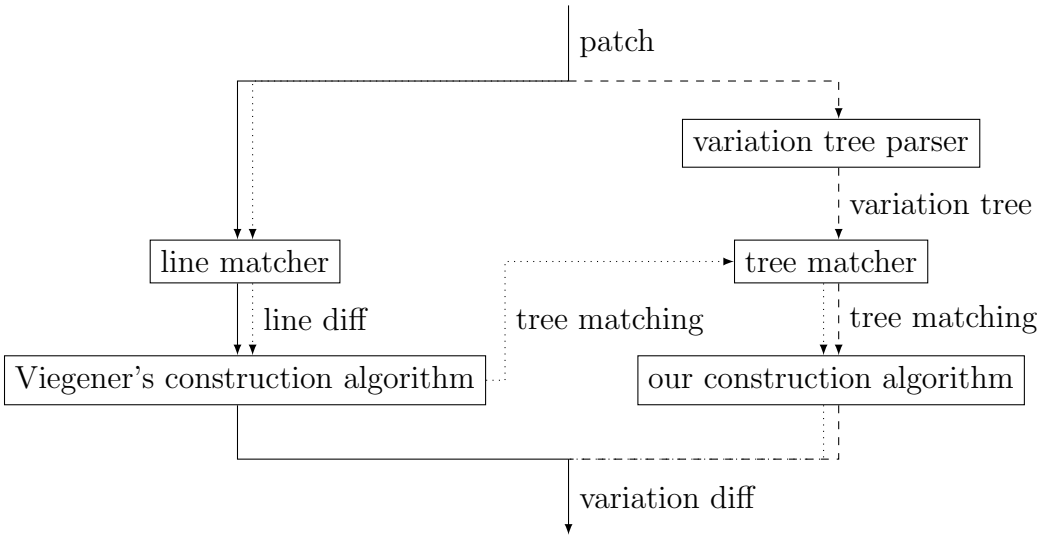


Figure 5.1: Dataflow between the construction phases

improvement procedure of the matching of variation diffs. It receives the first variation diff, constructed using a line matcher, as input, extracts the underlying matching, applies GumTree to this matching and reconstructs the variation diff using the new matching. This simplifies the implementation of the hybrid matcher to extracting the matching from a variation diff and some performance optimisations.

We separate the construction of each variation diff into a sequence of phases so we can compare the runtime of these phases individually. There are three distinct phases: variation tree parsing, matching, and variation diff construction. The dataflow between these phases can be seen in Figure 5.1. Each box represents an algorithm which are aligned in rows representing the phases. Variation tree parsing is only required for the second variation diff where the tree matcher needs concrete variation trees as input. In contrast, the variation trees of the other two variation diffs are constructed implicitly by Viegener's algorithm. The matching phase and construction phase is common to all constructed variation diffs. Notably, the hybrid approach runs the matching and construction phases twice, but not in order. In our implementation, there is a slight overhead for preparing variation trees and the matching of a variation diff into the input of GumTree. We account such overhead to the construction phases because, in our implementation, the matcher is called by the construction algorithm.

5.2 Datasets

As data source we use the Git history of the three following open-source software product lines:

Vim is a common UNIX editor written in C. The portability objectives of Vim require much platform dependent code which leads to an extensive usage of the C preprocessor.

	Vim	Busybox	Marlin	Sum
extracted commits	15354	17447	18570	51371
commits without relevant changes	892	2872	3980	7744
commits with invalid syntax	8	79	536	623
processed commits	14454	14496	14054	43004
total patches	39573	41546	80258	161377
patches with matching errors	13	27	26	66
processed patches	39560	41519	80232	161311
artifact nodes before the edit	189673878	47683365	81562821	318920064
annotation nodes before the edit	9876923	1368961	6107325	17353209

Table 5.1: Statistics of the dataset used for evaluation

Busybox is a minimal implementation of the core POSIX command line utilities written in C. It uses the C preprocessor to compile subsets of the provided features and utilities for optimizing embedded systems.

Marlin is a 3D printers driver written in C++ which receives commands (i.e. movements), reads sensors (i.e. temperature sensors) and controls actuators (i.e. step motors) in response. For supporting various different features, sensors, actuators and CPU architectures it uses the C preprocessor as well.

All analyzed software product lines use C or C++ for their implementation artifacts and use the C preprocessor for specifying the variability information of source code artifacts. Hence, we only analyze C and C++ source files which we identify by their file endings `.c`, `.h`, `.cpp` and `.hpp`. Table 5.1 shows the number of commits we extracted from the main branch of each repository. It is unclear for which parent we should analyze a change if a commit has multiple parents. Hence, we also filter out merge commits. Of the total 51371 extracted commits about 15.07% are merges or do not modify a C or C++ file. About 1.21% of the total commits contain some error in the preprocessor syntax. Examples of invalid preprocessor syntax include missing `#endifs`, missing formulas for `#ifs` and extra `#endifs`. In summary, we processed 43004 (83.71%) of the extracted commits which resulted in a total of 336273273 nodes in the variation trees before the edit.

5.3 Results

Table 5.1 shows that of the 161311 total patches, about 66 patches resulted in an invalid matching. These invalid matchings violate an implementation invariant and where all created by the hybrid matcher when applying GumTree to the line matching. The other two variation diffs have been constructed successfully but where omitted from all results to ensure comparability.

We present the average, median and accumulative runtime in Table 5.2 for each of the three constructed variation trees, distinguished by their matching algorithm. Other than the total time from extracting the edit from the Git history to the final variation diffs this table also shows the runtime of the individual phases. The biggest runtime difference exists between the matching algorithms where the tree differ took about 40000 times as

	matcher	variation tree parsing	matching	variation diff construction	total
average	line	-	0.63	4.60	5.23
	tree	10.59	25668.82	8.02	25676.84
	hybrid	-	4601.92	11.30	4613.22
median	line	-	0	2	2
	tree	6	40	2	98
	hybrid	-	26	5	69
accumulation	line	-	101290	741466	842756
	tree	1708282	4140663416	1292997	4143664695
	hybrid	-	742340370	1823435	744163805

Table 5.2: Average, median, and accumulative runtime in milliseconds of different phases of the construction algorithms

long as the line differ. Interestingly, the runtime of the tree differ is reduced to about one fifth when first calculating a line matching. In contrast to the drastic differences of the average runtime between the matching algorithms, the difference between their median is quite small. Parsing the variation trees takes about twice as long as constructing one variation diff with Viegener's algorithm. Compared to computing the matching, constructing the variation diffs is fast and the runtime difference between Viegener's algorithm and our algorithm very small. As expected, that the hybrid matcher takes about the same time as the line and tree matcher combined. Hence, the total construction runtime differences have the same magnitude as the matching runtime differences.

The tree matching and construction runtime behaviour in respect to the size of the input variation tree before the edit can be seen in Figure 5.2 and Figure 5.3. In Figure 5.2 there are two clusters, the cluster on the left seems to follow a polynomial (note that the scale is double logarithmic), whereas the second cluster is much more widespread. Significantly, there is a very sharp boundary between these two clusters at 1000 nodes in the variation tree before the edit. Similarly, the runtime of our variation tree construction algorithm can be seen in Figure 5.3. Apart from some outliers at the top left of the figure, most data points form a line with a hard boundary on the bottom and a smooth boundary with outliers to the top. The density of the data points in highest between around 0 and 15000 nodes in the variation tree before the edit. In contrast to Figure 5.2 there is no sharp edge in Figure 5.3 Hence, there might be some correlation between the size of the edited files with the runtime of the tree matcher which causes the tree matcher to behave differently.

The results of the simple quality metrics from Chapter 4 are show in Table 5.3. The matching size of the tree matcher is about 59.61% lower as the matching size of the line matcher. In contrast the matching size of the hybrid matcher increased by about 0.02% percent. As expected, the node count increased accordingly. Similarly, the edge count increased by about 59.12% between the line and tree matcher whereas it dropped by about 0.04% between the line and hybrid matcher. In fact, about 60.03% of the line and hybrid matchings are equal.

The flow of edit classes between the line and tree matcher can be seen in Figure 5.4a whereas the flow of edit classes between the line and hybrid matcher can be seen in

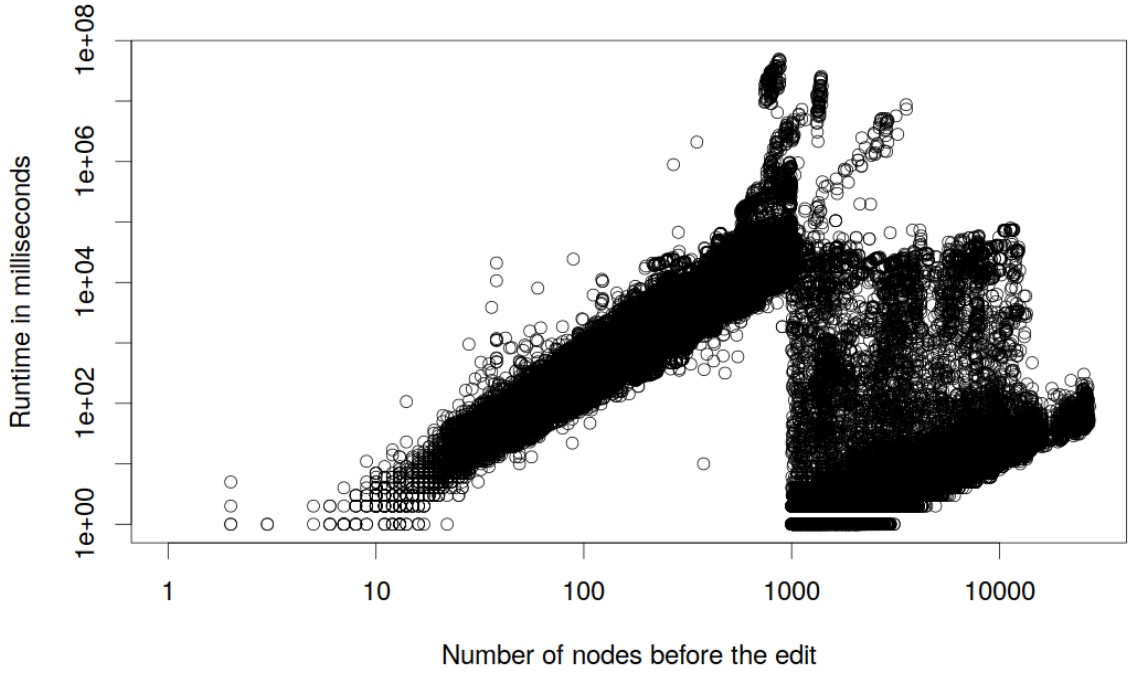


Figure 5.2: Runtime of the tree matcher (GumTree) depending on the size of the variation tree before the edit

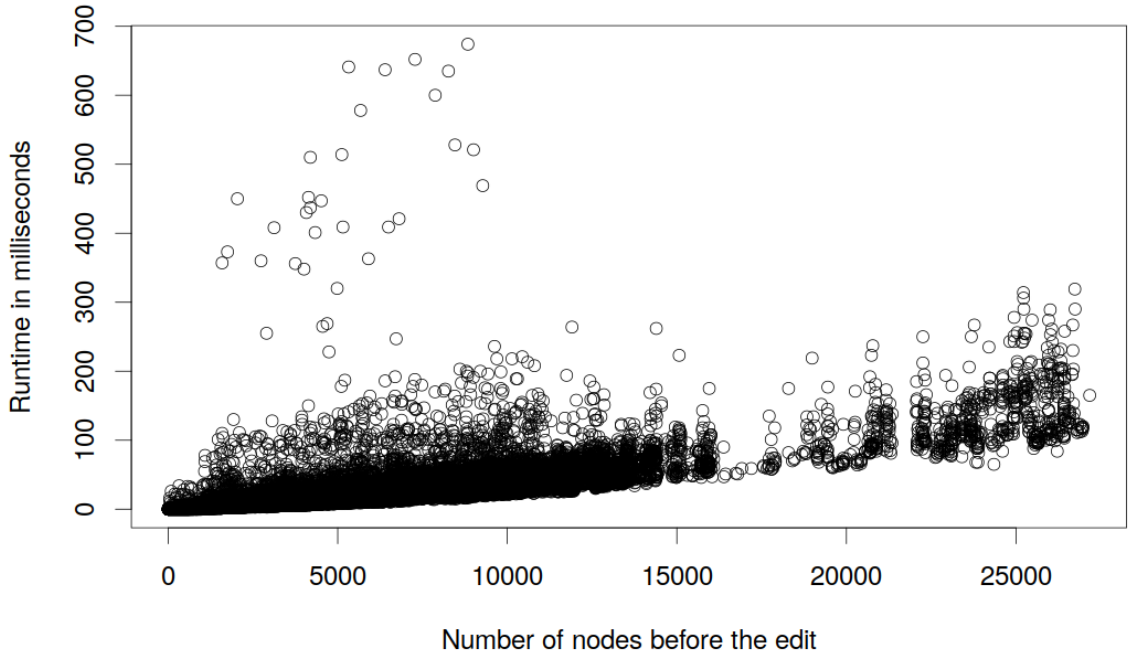
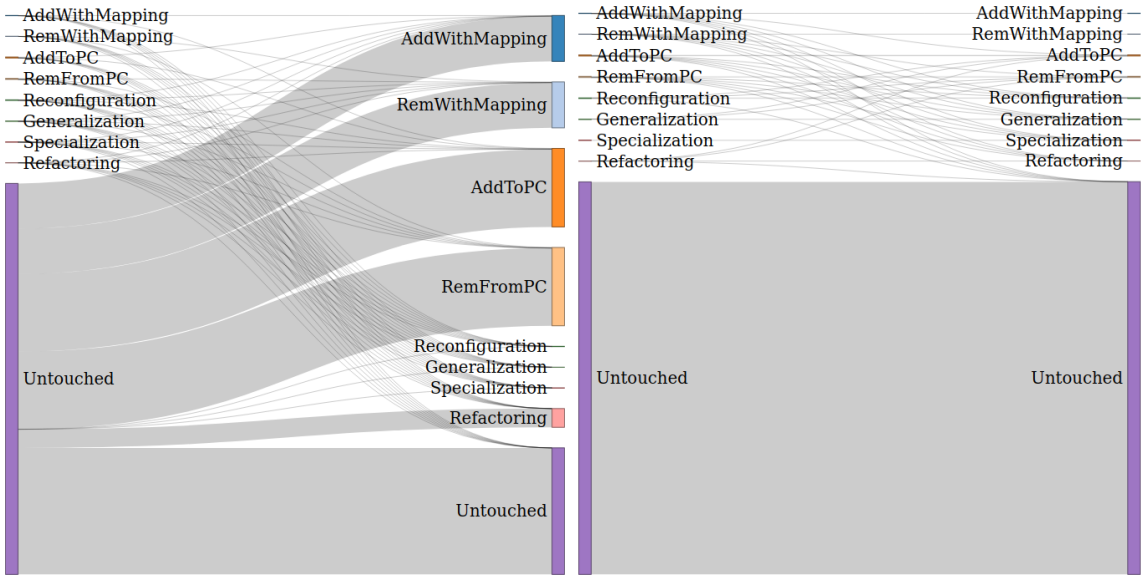


Figure 5.3: Runtime for constructing variation diffs using our constructing algorithm and a tree matching depending on the size of the variation tree before the edit

matcher	matching size	node count	edge count
line	334552494	338314332	338977945
tree	135111258	537755568	539396166
hybrid	334613928	338252898	338834085

Table 5.3: Simple metrics for the three variation diffs constructed using different matching algorithms



(a) Sankey-diagram of the flow of edit classes from the line matcher to the tree matcher

(b) Sankey-diagram of the flow of edit classes from the line matcher to the hybrid matcher

Figure 5.4b. Both figures are Sankey diagrams where the thickness of the line from the original edit class on the left to the new edit class on the right represents the number of edit classes flowing that particular path. To keep the amount of edit classes constant, unchanged artifacts are counted twice. For example, when a addition and deletion change into an unchanged node, that unchanged node is counted twice. With the line and hybrid matcher the vast majority of artifact nodes are classified as Untouched. In contrast most of the Untouched artifacts from the line differ change into different edit classes when matched by the tree matcher. The details of Figure 5.4b can be read in Table 5.4. Each row of that table represents one flow from the edit class in the first column to the edit class in the second column, thus it represents the exact same data as Figure 5.4b. The only change of edit classes with a flow rate bigger than a hundred thousand is from Refactoring to Untouched. Indeed the biggest overall change in edit class count is the increase of Untouched by 289246 nodes. The only flows to a more general edit class, according to our metric in Section 4.2.2, are from Generalization, Reconfiguration and Refactoring to AddToPC and RemFromPC. These flows from specific to more general edit classes account to about 0.01% of the total 0.09% changed edit classes.

5.4 Discussion

There were no assertion failures related to any invariant related to our construction algorithm. We found 66 patches where GumTree produced an invalid matching when

edit class by the line matcher	edit class by the hybrid matcher	count
AddToPC	AddToPC	1539872
AddToPC	Generalization	547
AddToPC	Reconfiguration	19
AddToPC	Refactoring	31
AddToPC	Specialization	89
AddToPC	Untouched	12528
AddWithMapping	AddToPC	12543
AddWithMapping	AddWithMapping	330919
AddWithMapping	Generalization	1208
AddWithMapping	Reconfiguration	854
AddWithMapping	Refactoring	2186
AddWithMapping	Specialization	959
AddWithMapping	Untouched	30630
Generalization	AddToPC	11
Generalization	Generalization	880724
Generalization	RemFromPC	11
Reconfiguration	AddToPC	5
Reconfiguration	Reconfiguration	756718
Reconfiguration	RemFromPC	5
Reconfiguration	Untouched	2
Refactoring	AddToPC	9
Refactoring	Refactoring	61028
Refactoring	RemFromPC	9
Refactoring	Untouched	202928
RemFromPC	Generalization	90
RemFromPC	Reconfiguration	44
RemFromPC	Refactoring	46
RemFromPC	RemFromPC	1288726
RemFromPC	Specialization	709
RemFromPC	Untouched	12528
RemWithMapping	Generalization	1665
RemWithMapping	Reconfiguration	829
RemWithMapping	Refactoring	2171
RemWithMapping	RemFromPC	14883
RemWithMapping	RemWithMapping	279045
RemWithMapping	Specialization	339
RemWithMapping	Untouched	30630
Specialization	Specialization	316530
Untouched	Untouched	632358738

Table 5.4: Flow of edit classes from the line matcher to the hybrid matcher

run as second matcher for the hybrid matching algorithm. Furthermore, there is a bug in either DiffDetective of JGit as there where 193 commits where DiffDetective reported a missing diff symbol. None of these issues is caused by our construction algorithm, hence we verified the consistency of our constructing algorithm (RG1). In all fully processed patches, the three constructed variation diffs where semantically equivalent, hence we also verified the correctness of our constructing algorithm (RG1).

5.4.1 Benchmark (RG2)

There are some measurement artifacts in the form of sharp lines at the bottom of Figure 5.2. These artifacts are are due to the logarithmic y-axis and the accuracy of our runtime measurements of 1 millisecond. Furthermore, there are some values with a measure runtime of 0 milliseconds, these are of course not actually instant, but fast enough to be subject to measurement imprecisions. As these artifacts are purely visual we can safely ignore them for the rest of our discussion.

The two clusters in Figure 5.2 is caused by a threshold in GumTree. Falleri et al. [16] call this threshold $maxSize$ and the GumTree library assigns this threshold the value 1000. It is used to avoid an algorithm with cubic runtime complexity in the case of big input trees. Our observation is compatible with this theory as the cluster below 1000 nodes in size indeed seems to grows polynomial (note the double logarithmic axis). Hence, the tree matcher switches between two algorithms to improve the runtime performance for big inputs.

Nonetheless, GumTree needs about 40879 as much time as the line differ for calculating all matchings. The average runtime of GumTree is about 25.66 seconds which is too slow for interactive use. Even for non-interactive use, the benefit of using GumTree is probably too small when compared to the average runtime of below 1 millisecond for computing a line matching. Note, that this comparison might be biased towards the line matcher because the line matcher implementation is provided by JGit which is heavily optimized for practical usage whereas GumTree and our tree structure are more research oriented implementations with less focus on runtime optimisations. The median of just 40 milliseconds for the runtime of GumTree shows that only a minority of the constructed variation diffs need excessive amounts of runtime. The runtime of the hybrid matcher is shorter than the tree matcher alone. The average runtime of the hybrid matcher is about 4.60 seconds which, with more optimisations, might be feasible for interactive use. This runtime reduction is caused by fully matched nodes where GumTree short-circuits because it cannot add any new matchings. Hence, it might be possible to adapt some thresholds for optimizing GumTree for interactive usage when taking advantage of pre-matched variation trees in the hybrid matcher.

Figure 5.3 suggests, that our variation diff construction algorithm runs in linear runtime for most data points. In average and in total it is about half as fast as Viegener's algorithm. A disadvantage of our approach is, that we have to parse all variation trees explicitly which even more costly than our construction algorithm by itself. In isolation, our algorithm scales linearly with the size of the input but when combined with GumTree the whole performance drops to an potentially impractical level. Although, it is possible to directly use a line matcher with our construction algorithm, our construction algorithm is slower and does not provide any benefit in comparison to Viegener's

algorithm when using a line matcher. In essence, our variation diff construction method is more flexible but about 4 times as slow as Viegener's algorithm and needs a more performant matching algorithm than GumTree for being practical.

5.4.2 Variation Diff Quality (RG3)

The matchings computed by GumTree are smaller and result in less specific edit classes than the matchings computed by the line differ. Figure 5.4a visualizes how many edit classes are matches with less specific edit classes instead. This is exactly the behaviour we want to avoid according to our edit class metric from Chapter 4 because it reduces the potential for meaningful results for analyses on variation diffs. We conclude that GumTree is not suitable for matching variation trees with line granularity.

In contrast, the hybrid matcher can only increase the size of a matching when compared to the matching of a line differ. Indeed, the matching size not only increases but almost all edit class flows result in more specific edit classes which can be seen in Table 5.4. More specific edit classes are beneficial for the interpretation of edit class analyses thus we interpret this as an improvement over the line matcher. Such improvements can be mainly attributed to detecting moved nodes because the line matcher we used cannot detect such moves. When taking advantage of the equivalence between variation diffs and matchings between variation trees as we did in our implementation it is useful to view the hybrid matcher as an improvement algorithm for variation diffs constructed using Viegener's algorithm. Hence, tree differs, such as GumTree, can be used to improve the quality of variation diffs for evaluating the edit classes of artifact nodes.

5.4.3 Threats to Validity

We only analyze three different software product lines, all of which use the C preprocessor for expressing variability. This selection could introduce a bias into the results. Expressing variability with the C preprocessor might favour some specific variation tree structures. For simplicity we limit our evaluation and conclusions to C preprocessor based software product lines although there are other sources of variability in the analyzed systems (e.g. Autoconf). Similarly, the development teams can organize their projects using different policies for their Git histories creating biases in their edit structures. For mitigation of these threats, we choose multiple widely studied open source projects.

The C preprocessor language has many construct which can be difficult to interpret correctly. Macros do not have a standard way of representing enabled and disabled features and are even capable of expressing C-like arithmetics. Hence, we apply boolean abstraction to simplify all C preprocessor expression to boolean formulas. This simplification is the state-of-the-art for checking different formulas for implications between.

Our quality metrics might not measure the quality of variation diffs as we expect. We explore each metric in detail in section Chapter 4 to ensure that we understand their benefits and weaknesses. The actual quality of variation diffs might still depend on specific applications. Hence, we developed a metric for the specific application of classifying edits to variability.

Although our construction algorithm is capable of constructing arbitrary granular variation diffs we only evaluated it on variation diffs with line granularity. For comparison

with the state-of-the-art construction algorithm which is not capable of generating arbitrary granular variation diffs, we have to evaluate our algorithm on line granularity too. It is reasonable that tree matchers do perform better on more granular input, because many tree differs are optimized for ASTs [13, 21]. Thus, we explicitly limit all of our conclusions to variation diffs with line granularity.

Our implementation or any dependency we use might have implementation bugs. To find any implementation bugs and answer RG1 we added many checks for internal consistency of most intermediate results. Using those checks, we found at least two bugs during our evaluation, one in GumTree and one in DiffDetective. These bugs are not caused by any logic relevant to our variation diff construction algorithm, thus they do not invalidate any of our results. Due to the code structure of DiffDetective we filtered whole commits if there where any parser errors but only patches if a consistency check failed (invalid matchings). As we parse all patches individually and dependencies between patches are not relevant for our metrics this has no meaningful influence on our results.

5.5 Summary

In this chapter, we verified the consistency and correctness of our construction algorithm and its implementation. While constructing variation diffs on three open-source software product lines two of our consistency and correctness checks reported errors. However, we these assertion failures where due to one bug in DiffDetective and one bug in GumTree when applied to line matchings. As these issues are independent of our construction algorithm, we successfully verified our construction algorithm.

The quality of the constructed variation diffs varied widely between the line matchings and tree matchings. According to all measured metrics, the tree matcher resulted in drastically worse variation diffs. In contrast, we found a qualitative improvement using the hybrid matcher. About 39.97% of the constructed variation diffs improved while the rest were equal to the variation diffs constructed using state-of-the-art algorithm.

The practicality of our algorithm depends mostly on the performance of the matching algorithm. Our tree matcher performed multiple orders of magnitude worse than the line matcher. Although the runtime improved when combining both matchers to the hybrid matcher, it may still be unpractical for many applications.

6. Related Work

In this chapter, we present research related to editing software product lines and diffing algorithms. In addition to this chapter, we discuss and reuse metrics used by line and tree differs in Section 4.2.1.

6.1 Variation Modeling

Bittner et al. [6] formalize the notion of variation trees and variation diffs to classify edit to software product lines. Viegener [44] uses these edit classes to empirically evaluate feature trace recording [5]. We use their edit classes as an application-specific metric for the quality of variation diffs. The original construction algorithm uses line-based diffs without move detection. In contrast, we utilize tree diffing algorithms to support arbitrary granular variation trees and recognize code movements which can improve the accuracy of classifications.

For developers, it can be useful to increase the granularity of feature annotations to reduce code replication and improve readability. Kästner et al. [23] demonstrate with *CIDE* that increase the granularity of feature annotations is practical by implementing an IDE which represents feature annotations as colors instead of preprocessor annotations. Furthermore, by increasing the overall granularity of the diffed source code it is possible to gain more precise diffs [16]. Our algorithm is capable of handling any granularity of inputs both in regards to feature annotations and code granularity by using a generic tree diffing algorithm.

6.2 Diffing Algorithms

There are various ways in which the quality of source code diffs can be improved. Line diffs as produced by the POSIX `diff` utility [11] are the most common and used by most version control systems such as Git. A common improvement to line-based diffing algorithms is the detection of moved lines [41]. In practice, whitespace often gets treated specially the reduce noise introduced, for example by indentation changes[1]. Word diffs

[1] For example, in Git, multiple adjacent whitespace characters are joined when passing the command line flag `--ignore-space-change`.

are another practical improvement which are achieved by splitting lines into a list of words and then diffing all words as if they were lines[2]. In contrast, tree differs compare the structure of source code by diffing abstract syntax trees instead of source code lines [17, 21, 31, 38]. Unlike variation trees, the results of all these algorithms do not include structured information about variability and cannot classify their effect on different variants. In contrast, our variation diffs provide the available variability and can be constructed using our variation diff construction algorithm in combination with a matching algorithm of one of the aforementioned diffing algorithms.

Moreover, diffing algorithms may be improved by exploiting domain specific knowledge. Vdiff [14] is a position-independent diffing algorithm for the Verilog language which exploits the concurrent semantics of Verilog. Apiwattanapong et al. [3] present *JDiff* which models object-oriented programs directly, thus resulting in more useful information for developers. A similar domain is targeted by *UMLDiff* [50]. In contrast to *JDiff*, *UMLDiff* provides information to design level changes instead of instruction level changes. All of these approaches have in common that they exploit or process additional information available in their domain to infer additional information. Variation diffs fit into the same category of domain specific diffing algorithms, but specialize in feature-based software product lines.

6.3 Analyzing Changes

Software product line evolution can introduce subtle defects [40] and erode the variability information over time [52]. Various methods exist to help developers understand changes to uncover such defects themselves or to catch defects automatically. By classifying edits to software product lines, a set of safe evolution templates can be derived [36]. Such templates provide invariants, for example that an edit does not modified the set of variants. As our variation diff construction algorithm can be used to classify edits done by developers, it could potentially be used to improve such templates. Furthermore, Gómez et al. [19] show that visualisation of diffs enables faster understanding of code changes. Variation diffs have the potential to provide such visualisations with information about the variability of code changes. Examples of such visualisation can be seen throughout this paper, for example in Figure 4.3.

In practice, changes to software product liens are checked for defects by developers reviewing code and automatic checks in continuous integration. Kästner et al. [25] created *TypeChef* which is able to parse annotated software product lines written in C and Java and check all variants for syntax [25] and type errors [28]. In contrast to checking for syntax errors after a change, it is also possible to guarantees syntactic correctness by construction CIDE [24]. Tests for software product lines might show defects in specific variant, but it is usually impractical to execute a test in all possible variants due to combinatorial explosion. Hence, a sample of all possible variants needs to be selected [37] or a symbolic execution for all variants has to be performed [26, 32, 46]. Variation diffs are a tool which could be used to reduce the overhead of such checks by analyzing what variants are affected. For example, a sample of variants to be tested could be reduced by the fact that not all variants are affect by a change.

[2]For example, in Git, word diffs can be enable using the command line flag `--word-diff`.

7. Conclusion

Variation diffs are a model for edits to the artifacts of a software product line. However, Viegener's algorithm, the state-of-the-art variation diff construction algorithm, cannot take advantage of the full flexibility variation diffs offer.

We present a general algorithm for constructing variation diffs from matchings between variation trees. In contrast to the state-of-the-art construction algorithm, our construction algorithm is able to fully take advantage of the flexibility variation diffs offer. Moreover, we prove a bijection between variation diffs and matchings between variation trees. Hence, differences between semantically equivalent variation diffs depend solely on the matching algorithm used during construction. An implementation of our variation diff construction algorithm is available in DiffDetective[1].

We identify four quality dimensions of variation diffs and provide metrics for comparing semantically equivalent variation diffs. In addition to general metrics, we present an application-specific metric which compares variation diffs according to the flow of edit classifications between them. By ordering edit classes according to their specificity, we can rate edit flows by their increase or decrease of specificity.

We evaluate the quality of variation diffs constructed using a line, tree and hybrid matcher on three open-source software product lines. In contrast to our matching algorithm agnostic construction algorithm, the state-of-the-art construction algorithm implicitly uses a line matcher, thus we use the line matcher as baseline for our comparison. The quality resulting from Gumtree, the tree matcher we use, is inferior to this baseline in respect to all metrics we measured. However, apart from 60.03% non-changing variation diffs, almost 39.97% of the evaluated variation diffs see a quality increase if Gumtree is employed in the hybrid approach of extending a line matching when compared to the baseline.

By benchmarking all variation diff constructions, we evaluate the scalability of our algorithm. The construction phase of our algorithm needs about double the time as the state-of-the-art algorithm and needs an additional variation tree parsing phase of the same order of magnitude of a few milliseconds. In comparison to the line matcher, the

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

tree and hybrid matchers are 5 orders of magnitudes slower. However, compared to the standalone tree matcher, the hybrid approach is about an order of magnitude faster although it runs the line and tree matcher internally. Hence, the bottleneck of constructing a variation diff using our algorithm is computing a matching between two variation trees.

In summary, our variation diff construction algorithm is capable of taking advantage of the full flexibility variation diffs offer and is able to compute higher quality variation diffs when combined with suitable matchers. However, the runtime of the evaluated tree and hybrid matchers might not be practical for many applications. Hence, we recommend to use Viegener's algorithm for line granular variation diffs and our construction algorithm if the increased flexibility or or the variation diff quality is more important than the construction runtime.

8. Future Work

In this section, we outline potential future work on the formal basis of variation diffs, quality metrics for variation diffs and the performance and quality of tree matching algorithms when applied to variation diffs.

We extend variation diffs with a children order. This children order allows us to model node movements below the same parent node which was previously impossible. However, in contrast to edit scripts, there is not enough information to uniquely decide which nodes were unmodified and which were moved if their parent didn't change. Deciding which node actually moved requires edit scripts for the children of unchanged nodes which enrich the children order with the required information. The structure of such edit scripts seems to be equivalent to line-based edit scripts with movement detection. A potential future extension could use line differs on the children of unchanged nodes.

It is currently unclear whether there is a generally useful quality metric for variation diffs. We hypothesize that such metrics are indeed application-specific but do not provide any proof of that. In the future, it might be necessary to refine and formalize further application-specific metrics. For example, the edit classification by Bittner et al. [6] used for our metric does not make use of the children order to identify movements. Instead, it classifies some moved artifacts as Untouched which might hide further optimization potential. Instead of manually interpreting the results of our metric, it could also be possible to algorithmically evaluate the edit flows, for example by assigning weights to different edit flows, analogous to how edit script costs are computed. Apart from such refinements, alternative metrics could be researched, for example applying metrics from graph theory to variation diffs. Then, different metrics should be compared to understand their differences and possibly extract a more general optimisation goal.

GumTree is, in comparison to the tested line matcher, slow. In the future, we should try to understand if this slowness is due to our specific inputs, the default thresholds of the GumTree library or if the slowness is inherent to the GumTree algorithm. Different tree differs may also be faster and result in better quality variation diffs. Hence, it is desirable to reevaluate our variation diff construction algorithm with different tree matchers and variation tree granularities.

Bibliography

[1] JTC 1/SC 22. *ISO/IEC 14882:2020*. ISO standard. International Organization for Standardization, 2020. URL: https://www.iso.org/standard/79358.html (cited on Page 5).

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Berlin, Heidelberg, Germany: Springer, 2013. ISBN: 978-3-642-37520-0. DOI: 10.1007/978-3-642-37521-7. URL: https://doi.org/10.1007/978-3-642-37521-7 (cited on Pages 1, 6).

[3] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. "A Differencing Algorithm for Object-Oriented Programs". In: *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. Washington, DC, USA: IEEE, 2004, pp. 2–13. DOI: 10.1109/ASE.2004.10015 (cited on Page 44).

[4] Philip Bille. "A Survey on Tree Edit Distance and Related Problems". In: *Theoretical Computer Science* 337.1–3 (2005), pp. 217–239. DOI: 10.1016/j.tcs.2004.12.030 (cited on Page 18).

[5] Paul Maximilian Bittner, Alexander Schultheiß, Thomas Thüm, Timo Kehrer, Jeffrey M. Young, and Lukas Linsbauer. "Feature Trace Recording". In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. Athens, Greece: ACM, Aug. 2021, pp. 1007–1020. ISBN: 9781450385626. DOI: 10.1145/3468264.3468531 (cited on Page 43).

[6] Paul Maximilian Bittner, Christof Tinnes, Alexander Schultheiß, Sören Viegener, Timo Kehrer, and Thomas Thüm. "Classifying Edits to Variability in Source Code". In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. Singapore, Singapore: ACM, Nov. 2022, pp. 196–208. ISBN: 9781450394130. DOI: 10.1145/3540250.3549108 (cited on Pages 1, 2, 6, 9–12, 29, 43, 47).

[7] Randal C Burns and Darrell D. E. Long. "A Linear Time, Constant Space Differencing Algorithm". In: *IEEE International Performance, Computing and Communications Conference*. IEEE. 1997, pp. 429–436 (cited on Page 27).

[8] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. "Identifying Changed Source Code Lines from Version Repositories". In: *Proc. Working Conf. on Mining Software Repositories (MSR)*. IEEE, May 2007, pp. 14–14. DOI: 10.1109/MSR.2007.14 (cited on Pages 2, 7).

[9] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. "Change Detection in Hierarchically Structured Information". In: *Proc. Int'l Conf. on Management of Data (SIGMOD)*. Montreal, QC, Canada: ACM, 1996, pp. 493–504. ISBN: 0-89791-794-4. DOI: 10.1145/233269.233366. URL: http://doi.acm.org/10.1145/233269.233366 (cited on Pages 2, 11, 18, 27, 32).

[10] Gregory Cobena, Serge Abiteboul, and Amelie Marian. "Detecting changes in XML documents". In: *Proceedings 18th International Conference on Data Engineering*. IEEE. 2002, pp. 41–52 (cited on Page 32).

[11] IEEE Portable Applications Standards Committee. *Portable Operating System Interface*. IEEE standard. Institute of Electrical and Electronics Engineers, 2017. URL: https://pubs.opengroup.org/onlinepubs/9699919799/ (cited on Pages 7, 43).

[12] Michael John Decker, Michael L. Collard, L. Gwenn Volkert, and Jonathan I. Maletic. "srcDiff: A Syntactic Differencing Approach to Improve the Understandability of Deltas". In: *J. Software: Evolution and Process* 32.4 (2020). DOI: 10.1002/smr.2226. URL: https://doi.org/10.1002/smr.2226 (cited on Pages 25, 27, 32).

[13] Georg Dotzler and Michael Philippsen. "Move-Optimized Source Code Tree Differencing". In: *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. Singapore, Singapore: ACM, 2016, pp. 660–671 (cited on Pages 2, 27, 32, 41).

[14] Adam Duley, Chris Spandikow, and Miryung Kim. "Vdiff: a program differencing algorithm for Verilog hardware description language". In: *Automated Software Engineering* 19.4 (2012), pp. 459–490. DOI: 10.1007/s10515-012-0107-6 (cited on Page 44).

[15] Martin Erwig and Eric Walkingshaw. "Variation Programming with the Choice Calculus". In: *Proc. Generative and Transformational Techniques in Software Engineering*. Berlin, Heidelberg, Germany: Springer, 2013, pp. 55–100. ISBN: 978-3-642-35991-0. DOI: 10.1007/978-3-642-35992-7_2 (cited on Pages 21, 22).

[16] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. "Fine-Grained and Accurate Source Code Differencing". In: *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. 2014, pp. 313–324. DOI: 10.1145/2642937.2642982. URL: http://doi.acm.org/10.1145/2642937.2642982 (cited on Pages 2, 25, 27, 32, 39, 43).

[17] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction". In: *IEEE Trans. on Software Engineering (TSE)* 33.11 (Nov. 2007), pp. 725–743. ISSN: 2326-3881. DOI: 10.1109/TSE.2007.70731 (cited on Pages 27, 32, 44).

[18] Veit Frick, Thomas Grassauer, Fabian Beck, and Martin Pinzger. "Generating Accurate and Compact Edit Scripts Using Tree Differencing". In: *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. Madrid, Spain: IEEE, Nov. 2018, pp. 264–274 (cited on Pages 27, 32).

[19] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo d'Hondt. "Visually characterizing source code changes". In: 98 (2015), pp. 376–393 (cited on Page 44).

[20] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. "On the Notion of Variability in Software Product Lines". In: *Proc. Working Conf. on Software Architecture (WICSA)*. 2001, pp. 45–54 (cited on Pages 1, 6).

[21] Masatomo Hashimoto and Akira Mori. "Diff/TS: A Tool for Fine-Grained Structural Change Analysis". In: *Proc. Working Conf. on Reverse Engineering (WCRE)*. 2008, pp. 279–288. DOI: 10.1109/WCRE.2008.44 (cited on Pages 27, 32, 41, 44).

[22] Paul Heckel. "A Technique for Isolating Differences between Files". In: *Comm. ACM* 21.4 (Apr. 1978), 264–268. ISSN: 0001-0782. DOI: 10.1145/359460.359467. URL: https://doi.org/10.1145/359460.359467 (cited on Pages 2, 7, 27).

[23] Christian Kästner, Sven Apel, and Martin Kuhlemann. "Granularity in Software Product Lines". In: *Proc. Int'l Conf. on Software Engineering (ICSE)*. Leipzig, Germany: ACM, May 2008, pp. 311–320. DOI: 10.1145/1368088.1368131 (cited on Page 43).

[24] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don Batory. "Guaranteeing Syntactic Correctness for All Product Line Variants: A Language-Independent Approach". In: *Proc. Int'l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE)*. Ed. by Manuel Oriol and Bertrand Meyer. Berlin, Heidelberg, Germany: Springer, 2009, pp. 175–194. ISBN: 978-3-642-02571-6. DOI: 10.1007/978-3-642-02571-6_11 (cited on Page 44).

[25] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. "Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation". In: *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. Portland, Oregon, USA: ACM, 2011, pp. 805–824. ISBN: 978-1-4503-0940-0. DOI: http://doi.acm.org/10.1145/2048066.2048128 (cited on Page 44).

[26] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. "Toward Variability-Aware Testing". In: *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*. Dresden, Germany: ACM, Sept. 2012, pp. 1–8. ISBN: 978-1-4503-1309-4. DOI: 10.1145/2377816.2377817 (cited on Page 44).

[27] Miryung Kim and David Notkin. "Discovering and Representing Systematic Code Changes". In: *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE. 2009, pp. 309–319 (cited on Page 27).

[28] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. "Scalable Analysis of Variable Software". In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. Saint Petersburg, Russia: ACM, Aug. 2013, pp. 81–91. ISBN: 978-1-4503-2237-9. DOI: 10.1145/2491411.2491437 (cited on Page 44).

[29] Tancred Lindholm, Jaakko Kangasharju, and Sasu Tarkoma. “Fast and Simple XML Tree Differencing by Sequence Alignment”. In: *Proc. of the ACM Symposium on Document Engineering*. 2006, pp. 75–84 (cited on Page 27).

[30] Jonathan I. Maletic and Michael L. Collard. “Supporting source code difference analysis”. In: *Proc. Int’l Conf. on Software Maintenance (ICSM)*. IEEE. 2004, pp. 210–219 (cited on Page 27).

[31] Junnosuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto. “Beyond GumTree: A Hybrid Approach to Generate Edit Scripts”. In: *Proc. Working Conf. on Mining Software Repositories (MSR)*. Montreal, QC, Canada: IEEE, 2019, pp. 550–554. DOI: 10.1109/MSR.2019.00082. URL: https://doi.org/10.1109/MSR.2019.00082 (cited on Pages 27, 32, 44).

[32] Jens Meinicke. “VarexJ: A Variability-Aware Interpreter for Java Applications”. Master’s Thesis. Germany: University of Magdeburg, Dec. 2014. URL: https://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/M14.pdf (cited on Page 44).

[33] Webb Miller and Eugene W. Myers. “A File Comparison Program”. In: *Software: Practice and Experience* 15.11 (1985), pp. 1025–1040. DOI: 10.1002/spe.4380151102 (cited on Page 27).

[34] Eugene W. Myers. “An O(ND) Difference Algorithm and Its Variations”. In: *Algorithmica* 1.1-4 (1986), pp. 251–266. DOI: 10.1007/BF01840446 (cited on Pages 2, 7, 27).

[35] Gonzalo Navarro. “A Guided Tour to Approximate String Matching”. In: *ACM Computing Surveys (CSUR)* 33.1 (2001), pp. 31–88 (cited on Pages 25, 27).

[36] Laís Neves, Paulo Borba, Vander Alves, Lucinéia Turnes, Leopoldo Teixeira, Demóstenes Sena, and Uirá Kulesza. “Safe Evolution Templates for Software Product Lines”. In: *J. Systems and Software (JSS)* 106 (2015), pp. 42–58. DOI: https://doi.org/10.1016/j.jss.2015.04.024. URL: https://www.sciencedirect.com/science/article/pii/S0164121215000801 (cited on Page 44).

[37] Sebastian Oster, Florian Markert, and Philipp Ritter. “Automated Incremental Pairwise Testing of Software Product Lines”. In: *Proc. Int’l Systems and Software Product Line Conf. (SPLC)*. Berlin, Heidelberg, Germany: Springer, 2010, pp. 196–210. DOI: http://dx.doi.org/10.1007/978-3-642-15579-6_14 (cited on Page 44).

[38] Mateusz Pawlik and Nikolaus Augsten. “RTED: A Robust Algorithm for the Tree Edit Distance”. In: *Computing Research Repository (CoRR)* 5.4 (2011), pp. 334–345. ISSN: 2150–8097. URL: http://arxiv.org/abs/1201.0230 (cited on Pages 2, 32, 44).

[39] Ian Sommerville. *Software Engineering*. 10th. Boston, MA, USA: Addison-Wesley, 2015. ISBN: 0133943038 (cited on Page 1).

[40] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. "Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem". In: *Proc. Europ. Conf. on Computer Systems (EuroSys)*. Salzburg, Austria: ACM, 2011, pp. 47–60. ISBN: 978-1-4503-0634-8. DOI: http://doi.acm.org/10.1145/1966445.1966451 (cited on Pages 1, 44).

[41] Walter F Tichy. "The string-to-string correction problem with block moves". In: *ACM Trans. on Computer Systems (TOCS)* 2.4 (1984), pp. 309–321 (cited on Page 43).

[42] Lauri Tischler and Bryan Henderson. *Linux Loadable Kernel Module HOWTO*. Accessed at December 29, 2022. Sept. 2006. URL: https://tldp.org/HOWTO/Module-HOWTO/index.html (cited on Page 1).

[43] Linus Torvalds. *Linux Operating System*. Accessed at December 29, 2022. URL: www.kernel.org (cited on Pages 1, 6).

[44] Sören Viegener. "Empirical Evaluation of Feature Trace Recording on the Edit History of Marlin". Bachelor's Thesis. Germany: University of Ulm, Apr. 2021. DOI: 10.18725/OPARU-38603. URL: https://oparu.uni-ulm.de/xmlui/bitstream/handle/123456789/38679/BA_Viegener.pdf (cited on Pages 2, 3, 18, 22, 32, 43).

[45] Eric Walkingshaw. "The Choice Calculus: A Formal Language of Variation". PhD thesis. USA: Oregon State University, June 2013. URL: https://ir.library.oregonstate.edu/concern/graduate_thesis_or_dissertations/k643b3668?locale=en (cited on Pages 21, 22).

[46] Chu-Pan Wong. "Beyond configurable systems: Applying variational execution to tackle large search spaces". PhD thesis. National University of Singapore, 2021 (cited on Page 44).

[47] X3J11. *ANSI X3.159-1989*. ANSI standard. American National Standards Institute, 1989. URL: https://webstore.ansi.org/standards/iso/isoiec98992018 (cited on Page 5).

[48] X3J11. *ISO/IEC 9899*. ISO standard. International Organization for Standardization, 1990. URL: https://www.iso.org/standard/17782.html (cited on Page 5).

[49] X3J11. *ISO/IEC 9899:2018*. ISO standard. International Organization for Standardization, 2018. URL: https://webstore.ansi.org/standards/iso/isoiec98992018 (cited on Page 5).

[50] Zhenchang Xing and Eleni Stroulia. "UMLDiff: An Algorithm for Object-Oriented Design Differencing". In: *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. New York, NY, USA: ACM, 2005, pp. 54–65. ISBN: 1581139934 (cited on Page 44).

[51] Chunhua Yang and E James Whitehead. "Pruning the ast with hunks to speed up tree differencing". In: *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2019, pp. 15–25 (cited on Page 32).

[52] Bo Zhang, Martin Becker, Thomas Patzke, Krzysztof Sierszecki, and Juha Erik Savolainen. "Variability evolution and erosion in industrial product lines: a case study". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Tokyo, Japan: ACM, Aug. 2013, pp. 168–177. ISBN: 978-1-4503-1968-3. DOI: 10.1145/2491627.2491645 (cited on Page 44).

Declaration of Authorship

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

Place, Date of Submission

Signature