Feature Trace Recording

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

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

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

Timo Kehrer
timo.kehrer@hu-berlin.de
Humboldt-University of Berlin
Berlin, Germany

Jeffrey M. Young
youngjef@oregonstate.edu
Oregon State University
Corvallis, USA

Lukas Linsbauer
l.linsbauer@tu-braunschweig.de
TU Braunschweig
Braunschweig, Germany

ABSTRACT

Tracing requirements to their implementation is crucial to all stakeholders of a software development process. When managing software variability, requirements are typically expressed in terms of features, a feature being a user-visible characteristic of the software. While feature traces are fully documented in software product lines, ad-hoc branching and forking, known as clone-and-own, is still the dominant way for developing multi-variant software systems in practice. Retroactive migration to product lines suffers from uncertainties and high effort because knowledge of feature traces must be recovered but is scattered across teams or even lost. We propose a semi-automated methodology for recording feature traces proactively, *during* software development when the necessary knowledge is present. To support the ongoing development of previously unmanaged clone-and-own projects, we explicitly deal with the absence of domain knowledge for both existing and new source code. We evaluate feature trace recording by replaying code edit patterns from the history of two real-world product lines. Our results show that feature trace recording reduces the manual effort to specify traces. Recorded feature traces could improve automation in change-propagation among cloned system variants and could reduce effort if developers decide to migrate to a product line.

CCS CONCEPTS

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

KEYWORDS

feature traceability, feature location, disciplined annotations, clone-and-own, software product lines

ACM Reference Format:
Paul Maximilian Bittner, Alexander Schultheiß, Thomas Thüm, Timo Kehrer, Jeffrey M. Young, and Lukas Linsbauer. 2021. Feature Trace Recording. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3468264.3468531

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468531

1 INTRODUCTION

For comprehending, maintaining, and extending existing software, it is crucial to find locations of interest in a software system quickly, reliably, and exhaustively [48, 89, 98]. To that end, tracing requirements to their implementation is one of the most common activities of developers and crucial to all stakeholders of a software development process [13, 35, 78, 83, 89, 98, 100]. When managing software variability, requirements are typically expressed in terms of *features*, a feature being informally defined as a user-visible characteristic of the software [12]. A *feature trace* identifies those artefacts of the software system that implement a certain feature, thus indicating where, how, and which features are implemented [8, 19].

While features, their dependencies, and their locations are fully documented in software product-line engineering [8, 19, 77], it is rarely adopted in practice. The reasons for poor adoption are unforeseen requirements at the beginning of development [57], a high up-front investment [16, 57], lack of tool support [93], missing flexibility [7, 84, 93], and necessary workflow adaptations [7].

In practice, development begins with only a single system variant of the software system to reduce complexity and costs, or because the need for future variants is unknown [7, 24, 57]. When the demand for a new variant emerges, a fast and easy approach is clone-and-own [7, 24, 84, 93]: By cloning the whole software system to alter specific parts independently from the previous variant (e.g., using branches or forks), developers can explore new ideas rapidly and without putting the actual system at risk [24, 93]. These cloned variants are meant to co-exist with the original variant to implement variability. Unfortunately, propagating changes such as bug fixes to other cloned variants is increasingly difficult and ambiguous with a growing number of variants [7, 24, 41, 51, 57, 84, 93]. Therefore, a considerable amount of research focuses on migrating clone-and-own software to product lines, allowing developers to switch to an integrated platform when managing a set of variants in parallel becomes infeasible [28, 40, 47, 50, 102]. However, migrations suffer from high uncertainties because knowledge of feature traces is scattered across the team or even lost [7, 24, 45, 51, 84, 85, 93]. The migration to a product line may even fail and thus bears considerable economical risks [28].

We propose *feature trace recording*, a methodology to infer feature traces semi-automatically upon source code changes. The main idea is that by recording feature traces during development, they neither have to be recovered retroactively, suffering from uncertainties, nor specified explicitly in a separate step, which causes

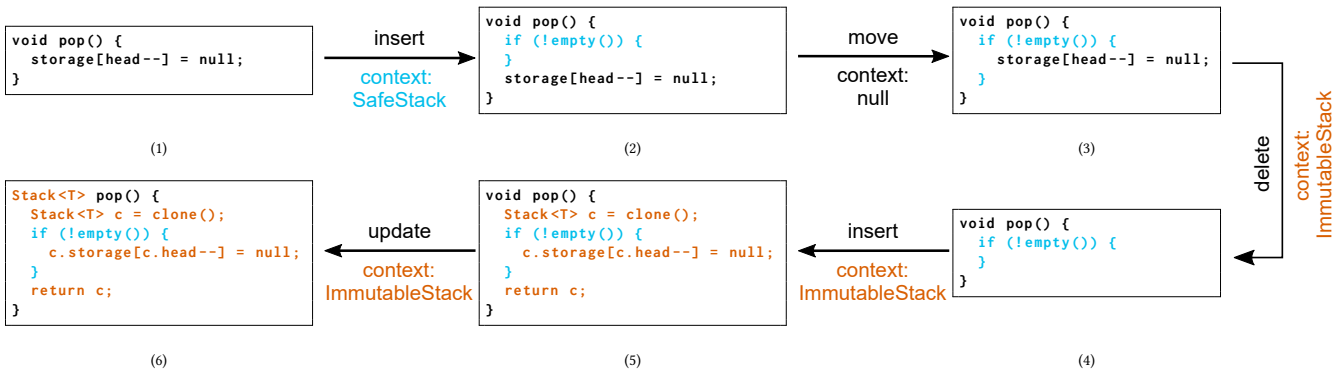

Figure 1: Feature traces are recorded upon edits from the feature context specified by the developer.

overheads to the actual source code editing and becomes increasingly difficult as time passes since the last edit. While Ji et al. showed with high manual effort that recording feature traces early can support teams in managing clone-and-own software [35], we are the first to propose a semi-automated methodology that inspects developer edits. Our core design philosophy is that contributing domain knowledge in the form of feature traces must place minimal burden on the user in order to be accepted. As developers might not always know to which feature an edited artefact belongs, and because feature traces are initially absent in clone-and-own development, we specifically deal with absent domain knowledge on both new and existing source code.

In our envisioned methodology, software development is performed according to a session-oriented editing model where developers *may* specify the feature or feature interaction they are currently implementing. From the edits developers make under such a *feature context* of an editing session, our recording algorithm infers feature traces for changed source code automatically. Moreover, by employing *disciplined annotations* by Kästner et al. [37] with Abstract Syntax Trees (ASTs) [38], we release developers from manual, laborious, and error-prone tasks, such as assigning opening and closing brackets to specify feature traces which are scattered among several code fragments. In summary, our contributions are:

Feature Trace Representation (Sec. 3). We generalize software product-line concepts to clone-and-own by explicitly encoding missing domain knowledge.

Feature Trace Recording (Sec. 4). We present a methodology for recording feature traces upon artefact changes from a possibly empty *feature context* specified by the developer.

Prototype and Evaluation (Sec. 5). We show that feature trace recording reduces manual effort to specify traces and enables evolving variability in clone-and-own at the level of edits common to software product lines.

2 OVERVIEW

Before formally introducing feature trace recording, we illustrate how developers can record feature traces, and show its potential impact by describing how recorded feature traces can (1) guide synchronisations in clone-and-own development and (2) be used as input for a migration to a software product line.

2.1 Motivating Example

Fig. 1 illustrates feature trace recording from the perspective of an imaginary Java developer called *Alice*, who is responsible for a single software variant (e.g., a branch or fork). Alice refines the pop method of a class `Stack`. We highlight source code implementing certain features with corresponding colours.

No feature traces are present in the initial code (1), illustrated by the absence of colours. Alice notices the pop method is unsafe; it does not check for an empty stack. She decides this error should only be prevented when defensive programming is desired, namely when the feature `SafeStack` is implemented. Alice makes this decision explicit by specifying `SafeStack` as the *feature context* of her edit (shown below arrows in Fig. 1). She inserts a condition to check whether the stack is empty, yielding snapshot (2). The inserted code is recorded as belonging to the feature `SafeStack`. After the insertion, Alice has to move the original implementation of the pop method into the condition. As Alice is not the original developer of the pop method, which does not exhibit any feature traces, she is unsure which feature the moved statement belongs to. By switching to the *empty* feature context (denoted as *null*), Alice can continue her working session without having to recover missing knowledge. In snapshot (3), the moved statement is not associated to any feature.

Later, Alice is assigned a new issue: *Stacks should be immutable*. She sets the feature context to `ImmutableStack` for the session and continues working on the pop method. First, she deletes the statement that modifies the stack's `storage`, leading to snapshot (4). Second, she inserts new code to implement the immutable variant (5) of pop. All inserted code is recorded to belong to the feature `ImmutableStack`, as specified by the feature context. Finally, the return type of pop has to be updated from `void` to `Stack<T>`, leading to snapshot (6). Feature trace recording assigns `ImmutableStack` to the updated return type.

In this example, Alice did not have to specify traceability information for each changed code fragment manually. Instead, she could record feature traces for almost the entire pop method from just three different feature contexts. The feature context is a concept we introduce for feature trace recording. It *may* be specified by developers to indicate the feature or feature interaction they are currently editing.

In general, feature trace recording is not tied to any specific task in the workflow of developers, such as commits to a version control

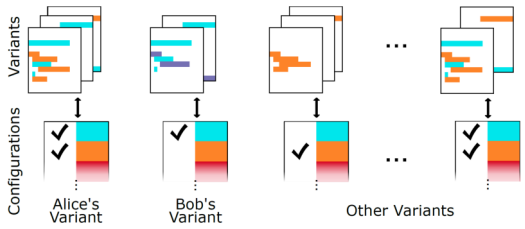

Figure 2: Feature traces recorded in several variants. Features are indicated by different colours.

system. Instead, we record feature traces at the granularity of *edits*, which we achieve with differencing techniques or logging editing commands as explained later in Sec. 4. In our example, Alice could commit to version control after her first two edits and at the end.

2.2 Scenario 1: Enhancing Clone-and-Own

In Fig. 2, we frame how feature trace recording could enhance clone-and-own development. By knowing feature traces (top row) and which variants implement those features (bottom row), target variants and code chunks for change synchronisation can be identified automatically. The only prerequisites are that developers agree on a common set of features [58, 83] and document which features are implemented by which variant [29, 57, 58]. This means, a cloned variant is equipped with a *configuration* [10, 19], stating for each feature whether it is implemented in this variant or not. In our example, Alice knows that her variant's configuration (bottom left) includes at least the two features `SafeStack` and `ImmutableStack`.

Imagine a second developer, *Bob*, wants to propagate Alice's changes from Fig. 1 into his variant, assuming that both variants are two of several cloned variants in a clone-and-own project. Bob's variant implements the feature `SafeStack` but not feature `ImmutableStack`. Further assume that both variants started with version (1) of `pop` in Fig. 1 and Bob did not apply simultaneous changes. Propagating Alice's edits to Bob's variant yields the implementation shown in Fig. 3. As `SafeStack` is implemented in Alice's *and* Bob's variant, propagating the first two edits from Fig. 1 will yield the same code as in snapshot (3). Subsequent edits on `ImmutableStack` are not synchronised as this feature is not present in Bob's variant.

Nonetheless, Alice *deleted* the statement `storage[head--] = null` from feature `ImmutableStack`. This means that the statement does **not** belong to `ImmutableStack` but it is still valid in those variants that do not implement `ImmutableStack` (e.g., Bob's variant). Thus, we recorded the feature trace `¬ImmutableStack` for the statement removed by Alice although it is not present in her variant any more. We do so by storing the history of edits such that we can recall where and when artefacts got deleted. The feature trace is then synchronised to Bob's variant in which the statement should still be present (purple colour in Fig. 3). The same applies for the return type `void` that was updated by Alice in her variant. Notably, the recorded trace `¬ImmutableStack` is different from Alice's feature context `ImmutableStack` as Alice's edit indicated that the deleted artefact does not belong to the edited feature.

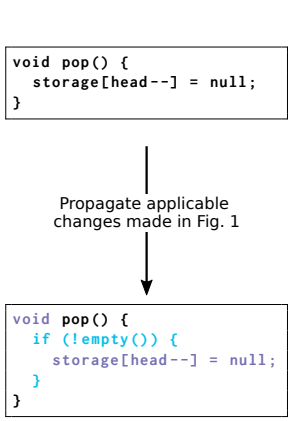

Figure 3: Propagating Alice's edits (see Fig. 1) to Bob's variant.

```
1  #ifdef ImmutableStack
2  Stack<T>
3  #else
4  void
5  #endif
6  pop() {
7  #ifdef ImmutableStack
8    Stack<T> c = clone();
9  #endif
10 #ifdef SafeStack
11   if (!empty()) {
12 #endif
13 #ifdef ImmutableStack
14     c.storage[c.head--]=null;
15 #else
16     storage[head--]=null;
17 #endif
18 #ifdef SafeStack
19   }
20 #endif
21 #ifdef ImmutableStack
22   return c;
23 #endif
24 }
```

Figure 4: The variants of Alice and Bob migrated to an integrated code base.

2.3 Scenario 2: Migrating to a Product Line

For migrating a set of cloned system variants into a software product line, feature traces are essential to extract corresponding source code to, for example, modules in a framework or an integrated code base. However, recovering feature traces in a post-mortem fashion (1) requires to halt development for an unknown duration and (2) is error-prone because knowledge on features is usually widespread across developers or even lost [7, 24, 45, 51, 84, 85, 93]. We argue that feature trace recording could help in (2) increasing migration accuracy as feature traces were recorded when developer's had the necessary knowledge, and (1) reduce migration time as less feature traces might have to be recovered.

Fig. 4 shows the `pop` method after migrating the variants of Alice and Bob to a preprocessor-based software product line. All variants are unified in a single code base where feature traces are specified *internally* (i.e., inside the source code) via `#ifdef`-directives. Thus, we omit colours as we use colours to represent feature traces that are stored *externally* (i.e., in a separate document). As shown in Fig. 1, Alice recorded traceability information for almost all code elements of `pop` by step (6). Hence, little effort is required to reach full traceability as only the feature of the definition of `pop` itself remains unknown (black colour in Fig. 1 and 3). For this example, we assign *true* to the definition of `pop`, meaning it is included in all variants and thus not surrounded by preprocessor directives in Fig. 4. Alice and Bob can retrieve their variants from Fig. 4 using their respective configurations from Fig. 2.

2.4 Our Contribution

In this paper, we focus on Alice's part of the example (see Sec. 2.1): formally introducing the methodology of feature trace recording and evaluating its applicability. The described scenarios of enhancing clone-and-own and migrations to product lines serve to illustrate potential benefits of recorded feature traces. We leave a more detailed investigation of the scenarios to future work for which feature traces are an essential prerequisite.

3 FEATURE TRACE REPRESENTATION

While most stakeholders are usually interested in feature traces (i.e., knowing which artefacts implement a given feature), from a technical point of view, it is more natural to consider feature mappings (i.e., which features are implemented by a given artefact). As we will later explain in more detail, feature traces can be inferred from feature mappings, and vice versa.

We do not store feature mappings by annotating source code directly with preprocessor macros (or similar annotations [35, 94]) for two reasons. First, preprocessor macros are known to be error-prone as they obfuscate source code, thus reducing readability and maintainability [37, 38, 52, 54, 63, 64, 91] as emphasized in Fig. 4. Second, we aim to maintain developers' existing workflows in which feature traces may not have been documented at all yet. To this end, we store feature mappings externally (i.e., in a separate file) and visualise them by colouring source code as shown in our motivating example in Fig. 1.

3.1 Minimizing Syntax Errors

In general, removing artefacts that implement the same feature should not invalidate the program, but should instead create a new variant. We refer to program invalidation in terms of violating a programming language's grammar as *syntax errors* [37]. Line-based feature traces, such as preprocessor macros, are unaware of the underlying language. It is the developer's responsibility to ensure that no syntax errors occur when deriving variants. For instance, in Fig. 4, the closing bracket in Line 19 has to be mapped to the same feature (SafeStack) as the opening bracket in Line 11. Feature traces that do not cause syntax errors upon removing associated implementation artefacts are referred to as *disciplined annotations* [39].

We transfer the product-line concept of disciplined annotations by Kästner et al. [37–39] to variant-oriented development, such as clone-and-own, by mapping features to nodes of an Abstract Syntax Tree (AST) instead of source code lines. ASTs describe the syntactic structure of an implementation artefact and are constructed from the grammar of the artefact's language by abstracting from concrete syntax [2]. In other words, an AST represents the structural elements of the corresponding program[1]. Fig. 5 shows the AST of version (3) of the pop method in our motivating example in Fig. 1.

When features are mapped to AST nodes, developers do not have to take care of syntactic elements such as commas or brackets. In Fig. 5, three nodes are assigned to the feature SafeStack, indicated by the solid blue colour; we describe the shapes and colours in detail below. To minimise potential syntax errors when removing a feature's nodes, we adhere to the following two rules introduced by Kästner et al. [39]:

Optional-Only Rule: Only *optional* nodes may be assigned to features [38]. For instance, entire methods can be safely removed from the AST. In contrast, the Statements node below the MethodDef shown in Fig. 5 is *mandatory* as removing it would invalidate the method. Though, mandatory nodes can still be removed when they are part of an optional subtree (e.g. removing the entire MethodDef subtree from its class). An AST's root is always optional as entire files, classes, or modules can always be removed. In Fig. 5, optional nodes have solid borders and mandatory nodes have dotted borders. All mandatory nodes are uncoloured or hatched as they are not mapped to any feature. Though, exceptions for some mandatory nodes are useful [38, 39]. For instance, we consider the return type of a method as optional as it can be implemented differently across variants (as in our motivating example in Sec. 2).

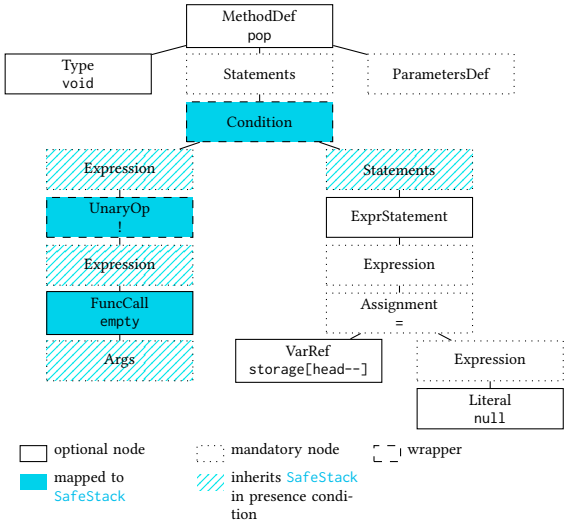

Figure 5: AST of version (3) of the pop method from Fig. 1.

Subtree Rule: ASTs directly unveil membership relations. For instance, statements that form the implementation of the pop method can be found in the subtree of pop's node in Fig. 5. As these statements cannot exist without the surrounding method, they should only be present when the method is also present. Thus, we propagate a node's feature mapping to all its descendants (i.e., all nodes in its subtree). An exception to this rule are *wrappers* [39] (dashed borders in Fig. 5): Constructs such as conditions, loops, or the UnaryOp in Fig. 5 that wrap nodes of the same type as the wrapper's parent can be removed safely without removing their wrapped content. For instance, removing the condition together with its expression subtree (left subtree in Fig. 5) but without its wrapped statements (right subtree) yields a valid AST again. Wrappers are always optional.

Formally, we write $v \in T$ if a node v is found in an AST T. Nodes can potentially be part of multiple ASTs, for instance the old and new version of an AST before and after a change by the developer.[2] In the next section, we describe how features are assigned to AST nodes and how we implement the optional-only and subtree rule.

3.2 Encoding Missing Knowledge

With feature trace recording, we support ongoing development of unmanaged clone-and-own projects in which artefacts may not be associated to features at all. For instance, Alice did not know the feature of the statement she moved, and feature knowledge on

[1] If no grammar is available (e.g., for a readme file), as a fallback solution, line breaks can be used to separate elements (i.e., lines of text).

[2] We employ differencing [17, 27, 30, 31, 75] or structural editing [65, 73, 80, 97] to determine if a node can be found in the new and old version of an AST (cf. Sec. 4.1).

the pop method and its return type was initially missing. Thus, the corresponding AST nodes are uncoloured in Fig. 5.

Formally, we introduce the propositional atomic value *null*. We treat *null* as the neutral element in negations and conjunctions (i.e., $\neg\, null \equiv null$ and $\varphi \wedge null \equiv \varphi$ for any propositional formula φ), yielding the ternary logic by Sobociński [81, p.70][90]. We refer to a propositional formula whose literals may have *null* as a possible value as *nullable propositional formula*. As software artefacts may be associated to single features, feature interactions, or even negations of features [8, 53, 59], we allow assigning nullable propositional formulas over features to AST nodes:

Definition 3.1. A feature mapping is a function $\mathcal{F}$ that maps each optional AST node to a nullable propositional formula over the set of features.

This definition of feature mappings respects the optional-only rule (cf. Sec. 3.1) as we only assign a formula to optional nodes. Also note that, from an AST with a corresponding feature mapping, one can acquire feature traces stating which artefacts implement a given feature f by grouping all nodes v where $f \models \mathcal{F}(v)$ (we write $A \models B$ if and only if $A \Rightarrow B$ is a tautology).

To account for the subtree rule, we introduce the *presence condition*. Given a feature mapping $\mathcal{F}$, the presence condition $\mathcal{PC}(T, \mathcal{F}, v)$ of a node $v \in T$ is a nullable propositional formula which states when v must be present, depending on its own mapping and its embedding into T:

$$\mathcal{PC}(T,\mathcal{F},v) := \begin{cases} \mathcal{F}(v) \wedge \bigwedge\limits_{a \in \mathcal{A}_T(v)} \mathcal{F}(a), & v \text{ is optional,} \\ \mathcal{PC}(T,\mathcal{F},p_T(v)), & \text{else.} \end{cases} \tag{1}$$

where $p_T(v)$ is the parent node of v in T and $\mathcal{A}_T(v)$ denotes the set of all optional non-wrapper ancestors of v in T. The presence condition of an optional node is a conjunction of its own feature mapping $\mathcal{F}(v)$ and the mappings of all its surrounding constructs (e.g., methods and classes). As mandatory nodes are not assigned to any feature and belong to their parent's definition, they have the same presence condition as their parent (the else case). In Fig. 5, solid and hatched blue nodes have the presence condition `SafeStack`. If a node $v \in T$ is unmapped (i.e., $\mathcal{F}(v) = null$), it neither invalidates nor contributes to its presence condition as *null* is neutral in conjunctions. For the same reason, a node mapped to *null* also does not influence the presence conditions of its descendants. Differentiating between feature mapping and presence condition separates two concerns: (1) assigning features to individual artefacts (AST nodes) and (2) respecting an artefact's structure (AST) to employ disciplined annotations. This allows us to only look at (1) feature mappings during recording in the next section while (2) presence conditions will ensure discipline automatically.

In summary, we represent feature traces as assignments of nullable propositional formulas to AST nodes (formally referred to as feature mappings) to account for missing knowledge on feature traces. With ASTs, we can minimize syntax errors when removing artefacts of a certain feature by implementing disciplined annotations [39] through the presence condition. We store feature mappings externally to support existing workflows in which features were probably not documented by developers yet and to avoid shortcomings of preprocessor annotations.

4 FEATURE TRACE RECORDING

Equipped with a formal specification for feature traces and artefacts, we now turn to their evolution by recording feature traces during development. We first formalise our notions of edits and feature context in Sec. 4.1, and present our recording algorithm in Sec. 4.2.

4.1 Editing of Development Artefacts

The goal of feature trace recording is to enable developers to document their domain knowledge during development. In accordance with our AST-based representation of feature traces, we handle software evolution on the level of ASTs. Specifically, we describe changes on ASTs as functions to which we refer as *edits*:

Definition 4.1. An edit e is a function, transforming an AST T to another AST $T' = e(T)$.

There are many types of edits [9, 14, 17, 27, 30, 31, 34, 70, 71, 75], where *insertion* and *deletion* can be considered primitives as they can model any change by deleting everything and inserting a new version. However, such an approach is infeasible as existing feature mappings may be lost upon deletion. Hence, we also consider *updates* and *moves*, where updates change properties of nodes, such as names, and moves relocate subtrees [14, 17, 27, 30, 31, 75].

As introduced in our motivating example in Sec. 2.1, we associate each edit with a feature context: a nullable propositional formula describing on which feature or feature interaction the developer is currently working. The feature context can be *null* as developers might sometimes not know to which feature an edited artefact belongs, such as Alice in her second edit in Fig. 1.

We restrict the feature context to formulas that are satisfiable in the current variant. This means, the feature context must be either *null* or satisfiable given the current variant's configuration (cf. Sec. 2.2). Thus, when developers specify the feature context, we ensure that they can only edit features comprised by their variant.

Detecting edits is possible through differencing [17, 27, 30, 31, 42, 43, 75] where a list of edits is derived from the old and new version of an AST (e.g., on commit to version control or document save). In structural editing [65, 73, 80, 97], used for domain-specific or graphical programming languages, user interactions with the IDE yield a list of edits directly. Given such a list of edits, we apply our algorithm sequentially to each edit separately.

Inspecting ASTs during development is a common mechanism for static analysis and code highlighting in modern IDEs, even on syntactically invalid states. Thus, we do not expect scalability issues when extending an IDE with feature trace recording because we can reuse the ASTs provided by the IDE during runtime. When adopting feature trace recording outside of an IDE (e.g., upon commit to version control), every changed file has to be parsed exactly once.

4.2 Feature Trace Recording Algorithm

Feature trace recording is defined in Alg. 1. It takes an edit e made to an AST T_{old} under feature context φ, and the existing feature mapping $\mathcal{F}_{old}$. It computes the feature mapping $\mathcal{F}_{new}$ for the new version of the AST $T_{new} = e(T_{old})$.

Feature trace recording is designed as a framework: For each type of edit t, it provides a dedicated recording function $\mathscr{R}_t$ handling edits of that type. Each $\mathscr{R}_t$ is a function that takes the old AST

Algorithm 1 Feature Trace Recording

Input: edit e, feature context φ, AST T_{old}, feature mapping $\mathcal{F}_{old}$
Output: new feature mapping $\mathcal{F}_{new}$

1: $t \leftarrow$ type of $e \in$ {ins, del, mov, up}
2: $\Delta \leftarrow$ AST nodes involved in e
3: $\mathcal{F}_{new} \leftarrow \mathscr{R}_t(T_{old}, \Delta, \mathcal{F}_{old}, \varphi)$ ▷ record new mapping
4: **return** $\mathcal{F}_{new}$

Table 1: Recorded feature mapping $\mathcal{F}_{new}(v)$ for edited AST nodes $v \in \Delta$ depending on the feature context φ and the old feature mapping $\mathcal{F}_{old}(v)$.

φ	$\mathcal{F}_{old}(v)$	$\mathscr{R}_{\text{ins}}$	$\mathscr{R}_{\text{del}}$	$\mathscr{R}_{\text{mov}}$	$\mathscr{R}_{\text{up}}$
defined	defined	-	$\mathcal{PC}(T_{old}, \mathcal{F}_{old}, v) \wedge \neg\varphi$	$\mathcal{F}_{old}(v) \wedge \varphi$	φ
defined	*null*	φ	$\neg\varphi$	φ	φ
null	defined	-	*false*	$\mathcal{F}_{old}(v)$	$\mathcal{F}_{old}(v)$
null	*null*	*null*	*null*	*null*	*null*

T_{old}, the set Δ of edited AST nodes, the old feature mapping $\mathcal{F}_{old}$, and the feature context φ to derive the next feature mapping $\mathcal{F}_{new}$ for T_{new}. Here, the edit e itself does not need to be considered in detail as we handle its type t by choosing a specialised recording function $\mathscr{R}_t$ and collecting the involved nodes in the set Δ. The set Δ comprises all inserted, removed, or updated nodes, depending on the edit being an insertion, deletion, or update, respectively. If the edit is a move, Δ is the set of all nodes in the moved subtree.

As a framework, feature trace recording can be tailored to specific needs by providing custom implementations for the recording functions $\mathscr{R}_t$. In this paper, we introduce recording functions for insertion, deletion, move, and update (i.e., $t \in$ {ins, del, mov, up}), but the general framework is open to be extended by functions that handle domain-specific edit operations (e.g., refactorings).

Table 1 provides an overview of the mapping $\mathcal{F}_{new}(v)$ produced by each of our recording functions $\mathscr{R}_t$ for edited nodes $v \in \Delta$. We discuss the construction of each recording function $\mathscr{R}_t$ in detail in the following section.

4.3 Recording Feature Traces for Specific Edits

In the following, we describe our implementation of the recording functions $\mathscr{R}_t$ for insertion, deletion, moves, and updates. According to Alg. 1, we determine feature mappings for all edited nodes $v \in \Delta$ from the previous mapping $\mathcal{F}_{old}$ in the old AST T_{old} and feature context φ, and keep mappings of unchanged nodes $v \notin \Delta$.

Insertions. We define the recording function $\mathscr{R}_{\text{ins}}$ that is used upon insertions as follows:

$$\mathscr{R}_{\text{ins}}(T_{old}, \Delta, \mathcal{F}_{old}, \varphi)(v) := \begin{cases} \varphi, & v \in \Delta, \\ \mathcal{F}_{old}(v), & \text{else.} \end{cases} \tag{2}$$

As inserted AST nodes $v \in \Delta$ are new to the code base, they do not have a feature mapping yet (i.e., $\mathcal{F}_{old}$ is undefined for them). The feature context φ describes exactly the feature or feature interaction developers are currently editing. Thus, we consider inserted nodes being *added* to that feature formula and assign φ to them. If developers do not specify a feature context, we do not know to which feature formula inserted nodes belong and thus assign *null*. Feature mappings of existing nodes $v \notin \Delta$ remain unchanged because they are not affected by insertions of other nodes.

Deletions. Deletions of artefacts $v \in \Delta$ should be propagated to exactly those variants that should not contain v anymore, as illustrated in Sec. 2.2. Thus, the major task for $\mathscr{R}_{\text{del}}$ is to differentiate between total deletions (removing an artefact from all variants) and partial deletions (removing an artefact in a particular feature interaction). We assign feature mappings to deleted artefacts for the sake of change synchronisation and to introduce the mappings to variants in which the artefacts are not deleted. We do so by storing the edit history to remember where and when artefacts were deleted, as pointed out in Sec. 2.2. For the remainder of this paper, we abbreviate the presence condition $\mathcal{PC}(T_{old}, \mathcal{F}_{old}, v)$ of a node v in the old version of the AST with pc_{old}.

$$\mathscr{R}_{\text{del}}(T_{old}, \Delta, \mathcal{F}_{old}, \varphi)(v) := \begin{cases} pc_{old} \wedge \neg\varphi, & v \in \Delta, \varphi \neq null, \\ false, & v \in \Delta, \varphi = null, pc_{old} \neq null, \\ null, & v \in \Delta, \varphi = null, pc_{old} = null, \\ \mathcal{F}_{old}(v), & v \notin \Delta\,. \end{cases} \tag{3}$$

We cover each case from top to bottom in a separate paragraph, except for the last case $v \notin \Delta$ which states that mappings of nodes being unaffected by the edit remain unchanged.

When developers delete a node v under a feature context $\varphi \neq null$, v does not belong to that feature formula anymore. However, v may still be a valid implementation in those variants not implementing the context φ but v's current feature formula pc_{old} as is shown in our motivating example in Fig. 1: Alice removed the statement `storage[head--] = null` from her variant to replace it with an implementation for `ImmutableStack`, but the statement was still a valid implementation in those variants not implementing `ImmutableStack` such as Bob's variant. Thus, deleting a node v under context φ introduces a feature interaction. We can refine its mapping by conjunction with $\neg\varphi$ and thereby remove v from exactly those variants containing the interaction $pc_{old} \wedge \varphi$. We reason on the presence condition pc_{old} of a deleted node v here instead of just its mapping $\mathcal{F}_{old}(v)$ because when removing v from the AST, we lose information on previous ancestors. Thus, we have to inline the presence condition here to remember existing dependencies. In case v does not have a presence condition, we assign $null \wedge \neg\varphi \equiv \neg\varphi$ to v as we only know that v should not be present in variants implementing φ anymore.

If the developer does not give information on which feature is currently edited explicitly (i.e., $\varphi = null$) but the deleted node $v \in \Delta$ has a presence condition $pc_{old} \neq null$, we know that v was a valid implementation in at least those variants whose configurations satisfy pc_{old}. We thus aim to remove v from all variants implementing pc_{old} and could assign $\neg pc_{old}$ to v. However, this would mean that v should be present even in those variants satisfying $\neg pc_{old}$ which did not contain v before the deletion and should not do so afterwards. Thus, there is no variant at all that should contain v after the deletion. We make this explicit by assigning *false* to v as *false* is unsatisfiable in all variants. In fact, a similar situation may also occur in the first case of $\mathscr{R}_{\text{del}}$, when the feature context φ is defined and equal to the deleted artefact's presence condition pc_{old}. In this case, v has to be removed from all variants because it should not be present in those variants implementing pc_{old} but should also not

be inserted to those variants not implementing pc_{old}. This is indeed the case because $pc_{old} \wedge \neg\varphi \equiv \varphi \wedge \neg\varphi \equiv false$ when $\varphi \equiv pc_{old}$ and $\varphi \neq null$. As before, assigning *false* to v is important because just $\neg\varphi$ would mean that v should be contained in those variants that did not contain v before the deletion.

When neither a feature context is given nor the deleted node $v \in \Delta$ is mapped (i.e., it's presence condition is *null*), we only know that it cannot be present in the current variant. As this does not give insight on which variants should contain v, we make this uncertainty explicit by assigning *null* to deleted artefacts.

Moves. The advantage of considering moves explicitly is the opportunity to keep existing mappings of moved nodes that would be lost when expressing moves in terms of deletion and insertion:

$$\mathscr{R}_{\text{mov}}(T_{old}, \Delta, \mathcal{F}_{old}, \varphi)(v) := \begin{cases} \mathcal{F}_{old}(v) \wedge \varphi, & v \in \Delta, \\ \mathcal{F}_{old}(v), & \text{else.} \end{cases} \quad (4)$$

Moving an artefact means extracting it from one place to locate it somewhere else, for instance, moving a method from one class to another. This means, the method does not belong to its previous class but to the new one. However, the method may be assigned a feature formula and also its statements may or may not implement several features. Thus, we decide to keep the mappings of moved artefacts (e.g., the method and its statements) but not the entire presence condition as developers decided to extract the artefacts from its surroundings (e.g., the class). Instead, presence conditions of moved artefacts adjust automatically according to surrounding constructs (i.e., previous and new ancestors in the AST). When developers indicate the currently edited feature formula in terms of the feature context φ, we know the move affects that feature or feature combination φ. Hence, we introduce a feature interaction $\mathcal{F}_{old}(v) \wedge \varphi$. When developers do not specify a feature context (i.e., $\varphi = null$), we keep the existing mapping without extending it because $\mathcal{F}_{old}(v) \wedge \varphi \equiv \mathcal{F}_{old}(v) \wedge null \equiv \mathcal{F}_{old}(v)$. This is what happened when Alice moved a statement in our motivating example in Sec. 2.1 without knowing which feature she is editing.

Updates. As for moves, considering updates explicitly allows to keep existing feature mappings that would be lost when expressing updates just in terms of insertions and deletions:

$$\mathscr{R}_{\text{up}}(T_{old}, \Delta, \mathcal{F}_{old}, \varphi)(v) := \begin{cases} \varphi, & \varphi \neq null, v \in \Delta, \\ \mathcal{F}_{old}(v), & \text{else.} \end{cases} \quad (5)$$

When a node $v \in \Delta$ is updated, we have to ensure that this update affects all variants implementing the currently edited feature φ. In contrast to moves, updates change the artefacts themselves and thus can be seen as more functional changes to the source code (e.g., when a function call is renamed and could point to a different function afterwards). Thus, we replace existing mappings of updated nodes $v \in \Delta$ if a feature context is given. For instance, in our motivating example in Sec. 2.1, Alice changed the return type of the pop method for the feature `ImmutableStack`. The mapping of the return type should be `ImmutableStack` after the update even if it would have been mapped to another feature previously. When no feature context is specified, we keep existing mappings as there is no evidence by the developer that the edited feature is not the feature of the updated artefact itself.

Summary. With Alg. 1, feature traces can be recorded upon edits to the software from the developer's feature context. When performing multiple edits on the same feature, the context can be kept and does not have to be switched as shown in Fig. 1. Feature trace recording can be employed on previously unmanaged code to gradually record knowledge on the code base as we explicitly deal with absent knowledge (cf. Sec. 3).

5 EVALUATION

By recording feature traces upon source code edits, developers can be released from three key challenges of documenting feature traces for multi-variant software systems: (1) Taking care of disciplined feature mappings, (2) dealing with the absence of domain knowledge, and (3) tracing artefacts that are deleted in the course of evolution. These are qualitative achievements over traditional source code annotations which are (1) known to be error-prone (see Sec. 3), (2) enforce an immediate decision on feature mappings, and (3) cannot handle the annotation of deleted artefacts.

Furthermore, we evaluate the applicability of feature trace recording by investigating whether it enables developers to edit variable software systems (RQ1). Recording should impose a low burden for developers (RQ2+3) and ideally be easier than specifying feature traces manually (RQ3).

RQ1 Does feature trace recording support code changes that are common when developing variable software?
RQ2 How often must the feature context be switched?
RQ3 How complex is the feature context in comparison to the desired feature trace formula?

5.1 Study Design

Ideally, feature trace recording would be evaluated by analysing a real-world history of edits with feature contexts. As we introduce the notion of a feature context in this paper, there are no existing datasets we could use for the sake of evaluation directly. Furthermore, feature traces are usually missing in existing clone-and-own projects or are imprecise when recovered retroactively [35] (e.g., on migrations as in Sec. 2.2). This impedes verifying the correctness of recorded traces as we are missing a ground truth to compare our results with. To the best of our knowledge, exact and complete feature traces can only be found in software product lines.

We thus inspect real-world edits to software product lines, as feature traces are known *before* and *after* edits. This lets us understand how feature traces have to change when developers edit source code. To this end, we analyse if and how an edit to a software product line can be turned into an edit on variants when employing feature trace recording (e.g., in clone-and-own). In terms of our motivating example, this means to inspect if Alice and Bob can reproduce possible edits to the unified pop method in Fig. 4 when they instead develop their respective variants with feature trace recording.

Therefore, we reuse the classification of edits to software product lines by Stănciulescu et al. [92]. They derived a set of *edit patterns*, that classify all edits in the history of the product line Marlin[3] and 99.27% of the edits in the product line Busybox[4]. Each pattern

[3] up to commit `3cfe1dce1` in https://github.com/MarlinFirmware/Marlin
[4] up to commit `a83e3ae` in https://git.busybox.net/busybox

describes a possible *kind* of edit to software artefacts and their feature mappings at a fine granularity (i.e., not on the level of commits but edits). In fact, a commit may consist of several pattern applications [92]. In our study, we analyse each edit pattern to see if feature trace recording is capable of reproducing respective changes to feature traces and which feature context is necessary.

5.2 Reproducing Edits to Software Product Lines in Clone-and-Own

In this section, we examine if and how edits to software product lines can be turned into edits when developing variants directly with feature trace recording (e.g., in clone-and-own). Table 2 gives an overview of the examined edit patterns. As feature trace recording works on code changes, we omit all patterns describing changes to feature traces only. Changing traces explicitly that way is orthogonal to our recording and can be provided by IDE tools. For each pattern, the number of matched edits in Marlin's commit history is given in the *#edits* column (computed by summing the #Multi and #Only columns in the original paper [92, p. 327 ff.]). We give the ratio of edits for each pattern in the *ratio* column. Feature mapping formulas that are to be inferred by a pattern are denoted by a propositional formula m. We use this target feature mapping formula m as our baseline to answer RQ2 and RQ3 because existing techniques for proactive feature tracing during development [35, 94] require specifying m for each artefact manually (cf., Sec. 6).

Each pattern is a possible kind of edit to source code and to the associated feature mappings. Thus, we depict feature mappings as preprocessor statements as in Fig. 4 when displaying the patterns. This enables us to show if and how feature mappings are altered when editing corresponding source code. Accordingly, preprocessor statements are *not* part of the edited code. Instead, they describe how feature mappings change upon an edit when turning a pattern to an edit on variants, just as preprocessor statements represent feature mappings in the migrated pop method in Fig. 4 but not in Alice, and Bob's variants of pop in Fig. 1 and Fig. 3.

Conditions could be expressed by any of `#ifdef`, `#ifndef`, or `#if`, of which we chose `#if` in the following as it is the most general. For visualising patterns, we adopt the *unified diff* notation. Added lines are green and labeled with +. Removed lines are pink and labeled with -. Lines without colour or marker remain unchanged.

We also simulate each pattern in our research prototype[5] with structural editing to see if derived feature contexts are indeed suitable to reproduce a pattern. Additionally, our prototype reproduces our motivating example from Fig. 1.

5.2.1 AddIfdef. The first edit pattern covers the insertion of source code with a feature mapping formula m:

```
+ #if m
+   /* inserted code */
+ #endif
```

In general, code is inserted and mapped to m. We can reproduce this pattern in any variant whose configuration is a satisfying assignment for the condition m. In such a variant, inserting code under feature context $\varphi := m$ would reproduce this pattern. Our algorithm uses $\mathscr{R}_{\text{ins}}$ and thereby assigns φ to the inserted code fragment. This happens in our motivating example in Sec. 2.1, when Alice inserts code under contexts `SafeStack` and `ImmutableStack`. *Inside an optional scope mapped to m*, (e.g., a class or method) the code can even be inserted without any context (i.e., *null*). It will inherit the outer scope's formula m in its presence condition (cf., Eq. 1). Thus, setting the context φ to *null* or any formula that is more general than m (i.e., $m \models \varphi$) is feasible. For instance, when $\varphi = m$, $\varphi = true$, or $\varphi = m \vee \ldots$, the presence condition of the code will be $m \wedge \varphi \equiv m$.

Table 2: Edit patterns with necessary feature contexts φ.

pattern name	#edits	ratio in %	target mapping	φ in general	φ inside respectively mapped scope	#variants to edit
AddIfdef	1098	9.0	m	m	*null* or $\preceq m$	1
AddIfdef*	456	3.8	$m_1, \ldots, m_n$	$m_{i_1}, \ldots, m_{i_j}$	$\forall m_{i_k}$: *null* or $\preceq m_{i_k}$	$\leq j$
AddIfdefElse	275	2.3	m	m	*null* or $\preceq m$	2
			$\neg m$	$\neg m$	*null* or $\preceq \neg m$	
AddIfdefWrapElse	60	0.5	$m, \neg m$	m	–	1
AddIfdefWrapThen	16	0.1	$m, \neg m$	$\neg m$	–	1
AddNormalCode	5554	45.7	m	m	*null* or $\preceq m$	1
RemNormalCode	4141	34.1	*false*	*true* or $\preceq pc_{old}$	*null* or $\preceq pc_{old}$	1
RemIfdef	558	4.6	*false*	*null* or $\preceq pc_{old}$	*null* or $\preceq pc_{old}$	≤ 2

where "$\preceq m$" means φ has to be equal to or weaker than m (i.e., $m \models \varphi$); and $1 \leq j \leq n$

5.2.2 AddIfdef.* This pattern groups n applications of the previous *AddIfdef* with conditions $m_1, \ldots, m_n$, $n \geq 2$. As some of the conditions might be identical, the feature context has to be changed at most n times. Thus, we have to repeat pattern *AddIfdef* j times, $1 \leq j \leq n$, with contexts $\varphi_k := m_{i_k}$, $k \in \{1, \ldots, j\}$, where each $i_k \in \{1, \ldots, n\}$ denotes a unique index, such that each condition is considered exactly once. When some conditions contradict each other (e.g., A and $\neg A$) or when there is no variant implementing all conditions at once, we have to add those condition's code to a different variant as we can only add code to a variant if it implements the desired formula φ_k. So in worst case, all j contexts contradict each other and thus the number of variants to edit is limited by j.

5.2.3 AddIfdefElse. Similar to *AddIfdef*, code surrounded by a condition is inserted but together with an `#else` branch:

```
+ #if m
+   /* inserted code (1) */
+ #else
+   /* inserted code (2) */
+ #endif
```

The two inserted code fragments are supposed to have the feature mappings $\mathcal{F}_{new}(1) = m$ and $\mathcal{F}_{new}(2) = \neg m$. We cannot reproduce this pattern directly in a single variant because both feature mappings are mutually exclusive (i.e., contradicting). Each fragment has to be added in a variant whose configuration satisfies the respective formula. *In general*, each code fragment has to be inserted with the corresponding feature context such that $\mathscr{R}_{\text{ins}}$ will be used to assign the context as mapping, similar to *AddIfdef*. Notably, the alternative code fragment (2) has to be inserted under $\varphi = \neg m$. This is possible, as we decided to use propositional formulas for feature context and mappings instead of simpler constructs such as lists of features employed elsewhere [29, 38]. *Inside an optional scope mapped to m or ¬m respectively*, inserting code under feature context *null* or φ with $m \models \varphi$ for code (1) and $\neg m \models \varphi$ for code (2) respectively, is sufficient, just as for *AddIfdef*.

[5] GitHub: https://github.com/pmbittner/FeatureTraceRecording/tree/esecfse21
DOI: 10.5281/zenodo.5109867

5.2.4 AddIfdefWrapElse. This pattern replaces an existing code fragment with an implementation for a certain feature:

```
+ #if m
+   /* inserted code (1) */
+ #else
    /* existing code (2) */
+ #endif
```

In general, reproducing this pattern is possible with just the single feature context $\varphi = m$ to record both mappings. We have to edit a single variant that implements m and is not supposed to contain the existing code (2) anymore. By deleting code (2) it will be mapped to $\neg m$ by $\mathscr{R}_{\text{del}}$. The new code (1) can then be inserted under the same feature context to map it to m with $\mathscr{R}_{\text{ins}}$. This is what Alice did in the motivating example with her third and fourth edit in Fig. 1: She sets the feature context to `ImmutableStack`, deletes `storage[head--] = null`, and replaces it with the immutable version afterwards. That way, Alice inserts new code mapped to `ImmutableStack` after deleting existing code that is recorded as belonging to `¬ImmutableStack` (i.e., the `else` case). *Inside an optional scope mapped to m*, this pattern would be ill-formed as it introduces dead code, even in the product line itself (e.g., Marlin): The else case (2) would get the presence condition $m \wedge \neg m \equiv \mathit{false}$.

5.2.5 AddIfdefWrapThen. Reciprocal to *AddIfdefWrapElse*, this pattern maps an existing artefact as belonging to a specific feature while introducing the more general case:

```
+ #if m
    /* existing code (1) */
+ #else
+   /* inserted code (2) */
+ #endif
```

In general, this pattern can be reproduced the same way as *AddIfdefWrapElse* but with the inverse feature context $\neg m$. In a variant containing the code (1) and satisfying $\neg m$, deleting (1) will let $\mathscr{R}_{\text{del}}$ determine $\neg\varphi \equiv \neg\neg m \equiv m$ as the new mapping. Subsequently, code (2) has to be inserted under the same context. Again, from a single feature context, two different feature mappings are recorded. *Inside an optional scope mapped to m*, this pattern is ill-formed even in a product line, just as for *AddIfdefWrapElse*.

5.2.6 AddNormalCode. This pattern comprises the insertion of code without any associated feature mapping. The inserted code is either non-variable or is supposed to exist inside other existing `#if` scopes. *In general*, the new code has to be inserted under feature context $\varphi = m$ being the target mapping m ($m = \mathit{true}$ for non-variable code) just as for *AddIfdef*. *Inside an optional scope mapped to m*, the new code can be inserted under feature context *null* or a formula weaker or equal to m just as for *AddIfdef* as the code will inherit the outer scope's formula in its presence condition.

5.2.7 RemNormalCode. This pattern depicts the removal of a code fragment, regardless of whether it is mapped to a feature or not:

```
  #if m
- /* removed code */
  #endif
```

As artefacts are removed from the entire software, they have to disappear from each variant containing them, which means the code has to be assigned *false*. *In general*, any feature context φ weaker than the deleted code's presence condition $pc_{old} \models \varphi$ (e.g., $\varphi = \mathit{true}$) is feasible. $\mathscr{R}_{\text{del}}$ will record $pc_{old} \wedge \neg\varphi \equiv \mathit{false}$ as both terms contradict each other (cf. Eq. 3). If $pc_{old} \equiv \mathit{null}$, the context φ has to be set to *true*. *Inside an optional scope mapped to m*, the feature context can even be set to *null* as $\mathscr{R}_{\text{del}}$ will assign *false*.

5.2.8 RemIfdef. This pattern comprises the removal of preprocessor blocks. It covers annotations with and without `#else` branch:

```
- #if m
-   /* removed code (1) */
- #else
-   /* removed code (2) */
- #endif
```

As for *RemNormalCode*, the removed code has to be assigned *false* as it is removed from the entire software. As for *AddIfdefElse*, both cases contradict each other, and thus two variants have to be edited. As the artefacts have a presence condition, deleting them without any feature context (i.e., *null*) is sufficient as $\mathscr{R}_{\text{del}}$ will record *false*. Analogous to *RemNormalCode*, any context φ with $pc_{old} \models \varphi$ works, as $\mathscr{R}_{\text{del}}$ will record $pc_{old} \wedge \neg\varphi \equiv \mathit{false}$ in this case.

Summary. We summarise the results of our investigation of the above patterns in Table 2. The possible feature contexts to reproduce each pattern and the number of variants that have to be edited are given in the respective columns. All patterns can be reproduced. When a pattern is applied inside a scope mapped to the target mapping, even *null* or multiple formulas are sufficient to record desired traces. If multiple variants have to be edited, this is due to contradictory mappings.

5.3 Discussion

RQ1 – Applicability. We showed that feature trace recording can be applied successfully to reproduce all presented edit patterns. We also reproduce all patterns in our prototype (cf. Sec. 5.2).

> Feature traces can be recorded for all considered edit patterns.

RQ2 – Feature Context Switches. We consider how many variants have to be edited, and how often a new feature context has to be set to reproduce each pattern.

Different variants have to be edited when feature formulas contradict each other or when no variant is compatible to all formulas at the same time. We believe editing multiple variants at once to be unlikely for clone-and-own development. Even in Marlin, artefacts with contradicting feature mappings were added or removed less often than editing just a single mapping or variant: Only 2.3% of the edits were classified by *AddIfdefElse*, 4.6% by *RemIfdef*, and 3.8% by *AddIfdef**, where the latter two may contain cases without conflicting feature mappings.

For all patterns, the amount of required feature context switches is equal to or smaller than the number of different target mappings, as shown in Table 2. A single feature context could also be reused when applying several patterns in a row as done by Alice in Sec. 2.1.

> For reproducing single patterns, user input is required less or as frequent as for directly specifying mappings. Feature context switches are potentially less frequent when applying multiple patterns in a row. In at least 89.3% of the cases only one variant has to be edited.

RQ3 – Feature Context Complexity. To answer this question, we compare the feature contexts with the target feature mappings for each pattern, summarised in Table 2.

All insertion related patterns (i.e., *Add...*), except for *AddIfdefWrapThen*, can be reproduced in our clone-and-own scenario by setting the feature context to exactly the desired feature mapping of the artefact to edit. When deleting artefacts, developers can differentiate whether they want to replace an artefact (cf. *AddIfdefWrapElse*, *AddIfdefWrapThen*) from a certain feature formula, or want to delete it from all variants (cf., *RemIfdef*, *RemNormalCode*), by specifying a corresponding feature context. In case of uncertainty, deleting an artefact v under a feature context φ is always a valid option because its refined mapping $pc_{old} \wedge \neg \varphi$ shrinks the set of variants containing v. This way, developers can iteratively decide whether v has to be removed from further variants (e.g., variants they are responsible for) such that v may indeed be removed from every variant eventually.

For all patterns, including the most common ones *AddNormalCode* and *RemNormalCode* that comprise 79,74% of all edits, the feature context may even be omitted. When artefacts are inserted or deleted inside scopes (e.g., a class or method) that already have the target mapping, *null* or several formulas are feasible. Only, the rare patterns *AddIfdefWrapThen* and *AddIfdefWrapElse* (0.62%) are ill-formed in this case as they introduce dead code when performed in the product line already.

> In general, the feature context has to be equal to the target feature mapping. For two patterns, different mappings can be recorded with just a single context. For more than 99% of the edits, *null* or a variety of formulas are a feasible context when an outer scope is already assigned to the target mapping.

5.4 Threats to Validity

Internal Validity. A possible bias could be introduced the way we determined the feature context. If multiple possibilities for reproducing a pattern exist, we present the simplest one we could identify in terms of number of edits and divergence of the feature context from the targeted feature mapping formula. In practice, developers could choose more complicated contexts but they could also do so for direct annotations.

Stănciulescu et al. admit that they might have introduced bias when identifying the patterns [92]. They iteratively identified regular expressions until all edits in the considered history of Marlin were classified and cross-validated their results with the product-line Busybox.

External Validity. The considered evolution might not be representative for clone-and-own development. As discussed in our study design in Sec. 5.1, software product lines turned out to be the only feasible datasets for our evaluation because clone-and-own software suffers from absent or imprecise feature traces. However, even software product lines lack the direct availability of feature contexts. We thus decided against investigating development histories of existing projects manually. Instead, we argue that reverse-engineering feature contexts for *possible* changes to product lines is the best trade-off for our evaluation.

The considered edit patterns might be incomplete regarding the evolution of variable software systems. The history of Marlin analysed by Stănciulescu et al. [92] contains 3,747 commits (excluding merge commits) that lead to ~40k lines of code with more than 140 features in 187 source files (without files for Arduino) in the last version at commit 3cfe1dce1. Furthermore, some patterns were already identified in previous work [21, 35, 74] increasing our confidence that the most common and essential patterns are included.

6 RELATED WORK

Feature trace recording is the first step towards our vision for bridging the gap between clone-and-own and software product lines [44], depicted in Sec. 2.2. To this end, we extend early ideas on our *VariantSync* framework [76] with the first structured and formal method for inferring feature traces from developers' edits. Preliminary results and ideas on feature trace recording were presented in the first author's master's thesis [15].

Feature trace recording is inspired by Ji et al. [35]. In an empirical study, they showed the benefits of recording traces in a line-based annotative approach known as *embedded annotations* [6, 26, 35] or *feature tags* [33]. Ji et al. simulated the development of variable software by manually inspecting the commits of the history of a clone-and-own project. For each commit, the first author embeds line-based feature traces into the source code via comments *"based on his understanding of the codebase and the change history"* [35, p. 63]. Thus, Ji et al. discovered insights on the effectiveness of documenting feature traces during development by embedding annotations manually. Sulír et al. [94] report similar results when using Java annotations for manually documenting concerns. Both studies required high manual effort. With feature trace recording, we introduce a structured and formalized methodology reducing this manual effort. Heidenreich et al. proposed FeatureMapper [32], a tool for the specification of feature mappings in model-based software product lines. While in recording mode, FeatureMapper assigns a pre-defined feature expression to developer edits. With feature trace recording, we do not plainly assign the feature context to edits and edited artefacts but derive formulas based on existing feature mappings. Moreover, FeatureMapper is a pure software product-line tool, which does not consider individual variants as necessary to support clone-and-own. In contrast, we explicitly inspect edits for information relevant to other variants (e.g., upon deletions, moves, and updates).

Feature location and recovery [22, 83], also known as *variability mining*, retroactively recovers or recommends [1] feature traces with static [3, 40, 102] or dynamic analyses [25, 99, 101], through comparing software variants [58, 62, 103], mining software repositories [4], or by combining these methods [36, 67]. With feature trace recording, we unlock software evolution as a new source for feature location. One common application of feature location and recovery is the migration to a software product line [28, 47, 50] as exemplified in Sec. 2.3. Feature trace recovery tools require numerous developer decisions on existing code [28, 29, 40, 45, 58, 62, 84, 104] for which the necessary domain knowledge may be lost. Fully automatic migration techniques [29, 58, 102, 104] cannot yet retrieve feature traces in sufficient quality [45, 85]. With feature trace recording, developers can document feature traces gradually whenever

they have the necessary knowledge. Thus, potentially less feature traces might have to be recovered when migrating as discussed in Sec. 2.3. To this end, we explicitly deal with the absence of domain knowledge. Recently, Michelon et al. raised awareness to distinguish features and their revisions in feature location tasks [68, 69]. While not modelled explicitly in this paper, feature trace recording supports recording feature revisions: Whenever a sequence of edits is performed under the same feature context, we can increment the version of all features that occur in at least one positive literal in the context's conjunctive normal form. More generally, feature trace recording may serve as one of the building blocks for a better integration and management of software evolution in both space and time, a major challenge which has recently gained increasing research interest across different research communities [5, 11, 95].

Inspired by the *product line tool* CIDE [38], we employ *disciplined annotations* [37, 39] to preserve syntactical correctness of feature mappings. Experiments conducted on discipline [54, 61, 87] reveal the benefits for such constraints in industrial practice. We implemented disciplined annotations in terms of presence conditions and extended them to be able to handle missing domain knowledge by using the ternary logic by Sobociński [81, p.70][90]. Classifying AST nodes by the underlying grammar as optional, mandatory, or wrapper nodes (see Sec. 3) was automated by Kästner et al. [39].

Variation control systems [18, 29, 55, 56, 92] and filtered product lines [88] deal with variability on the level of features but where software product lines are developed by editing (partial) variants. Variation control systems either simplify development of software product lines [92] or enhance clone-and-own development [29] and usually impose a transaction-based checkout-change-commit workflow. In contrast, feature trace recording is more general as it does not require but could be adopted in such a workflow. Thus, variation control systems can integrate feature trace recording when developers are submitting their changes to a (partial) variant back to the repository. In this regard, we directly solve an issue Stănciulescu et al. stated as future work for their system: *"How to handle the cases when an ambition is weaker than the projection?"* [92, p. 331], where *ambition* and *projection* are user-specified propositional formulas. With the *projection*, users check out a partial variant of a product line. When submitting changes to their system's projectional editor, the *ambition* describes which feature formula was edited. Stănciulescu et al. argue that a weaker ambition could be desirable when an edit should not only affect the edited projection (i.e., variant) but other variants as well, for example when fixing a bug [92, p. 331]. We directly address this question as in our scenario the feature context (similar to ambition) is always weaker than the configuration (similar to the projection).

Clone management in terms of clone detection [66, 84], elimination [82], tracking [20, 23, 72, 86, 96], or prevention [49], typically considers code clones as a small-scale phenomenon [46, 79]. However, we envision to enhance large-scale clone-and-own development at any stage of development. As we do not want to impair developers' habits and workflows significantly, we aim to enhance clone-and-own development instead of preventing it. Recently, Mahmood et al. proposed Virtual Platform, a method and tool for gradual variability management at different stages in the spectrum between pure clone-and-own and software product lines [60]. Just as variation control systems and migration methods, Virtual Platform also requires feature mappings which have to be specified manually by developers with the `MapAssetToFeature` operator. Feature trace recording could be a possible implementation for `MapAssetToFeature`.

7 CONCLUSION

Tracing features to their implementation is one of the most common tasks for developers and an essential prerequisite for variability management solutions. If feature traces are not documented proactively, as in software product lines, they have to be recovered retroactively which is often inaccurate or not possible for all traces. In turn, product lines come with a high up-front investment that may not pay off when introducing just a single new variant.

For this reason, we propose to record feature traces during software development. *Feature trace recording* infers feature traces upon source code edits from a feature context: a user-specified formula. While product-line techniques and variation control systems require complete feature traces and migration techniques fail to completely recover them, feature trace recording accounts for absent knowledge in both edited and existing source code.

In the future, we plan to employ recorded feature traces for variant synchronisation in clone-and-own as illustrated in Bob's part of our motivating example in Sec. 2.2. To this end, we aim at broadcasting edits and feature traces to all relevant code locations across cloned variants similar to cherry-picking. Variants might even become reconfigurable, when a certain degree of feature traces is accumulated. For all these ideas, the first and foremost requirement is the automated recording of feature traces.

ACKNOWLEDGMENT

We thank our reviewers for their constructive feedback. We also thank Duc Anh Vu, Alexander Boll, Helge Wrede, and Katharina Juhnke for proof reading; Tobias Heß and Pascal Kohlhepp for their help with the artefact; Tobias Pett and Ina Schaefer for early feedback; and Larissa Förster, Susana Castillo Alejandre, Marc Kassubeck, Moritz Kappel, Jann-Ole Henningson, Sascha Fricke, and Jan-Philipp Tauscher. This work has been partially supported by the German Research Foundation within the project *VariantSync* (TH 2387/1-1 and KE 2267/1-1).

REFERENCES

[1] Hadil Abukwaik, Andreas Burger, Berima Kweku Andam, and Thorsten Berger. 2018. Semi-Automated Feature Traceability with Embedded Annotations. In *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 529–533. https://doi.org/10.1109/ICSME.2018.00049

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley.

[3] Ra'Fat Al-Msie'deen, Abdelhak Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. 2013. Feature Location in a Collection of Software Product Variants Using Formal Concept Analysis. In *Proc. Int'l Conf. on Software Reuse (ICSR)*, John M. Favaro and Maurizio Morisio (Eds.), Vol. 7925. Springer, 302–307. https://doi.org/10.1007/978-3-642-38977-1_22

[4] Nasir Ali, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2013. Trustrace: Mining Software Repositories to Improve the Accuracy of Requirement Traceability Links. *IEEE Trans. on Software Engineering (TSE)* 39, 5 (2013), 725–741. https://doi.org/10.1109/TSE.2012.71

[5] Sofia Ananieva, Sandra Greiner, Thomas Kühn, Jacob Krüger, Lukas Linsbauer, Sten Grüner, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, Sebastian Krieter, Christoph Seidl, S. Ramesh, Ralf Reussner, and Bernhard Westfechtel. 2020. A Conceptual Model for Unifying Variability in Space and Time. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, Article 15, 12 pages. https://doi.org/10.1145/3382025.3414955

[6] Berima Andam, Andreas Burger, Thorsten Berger, and Michel R. V. Chaudron. 2017. FLOrIDA: Feature LOcatIon DAshboard for Extracting and Visualizing Feature Traces. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 100–107. https://doi.org/10.1145/3023956.3023967

[7] MichałAntkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Stefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. 2014. Flexible Product Line Engineering with a Virtual Platform. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 532–535. https://doi.org/10.1145/2591062.2591126

[8] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer. https://doi.org/10.1007/978-3-642-37521-7

[9] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Massimiliano Di Penta. 2013. LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines. In *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE, 230–239. https://doi.org/10.1109/ICSM.2013.34

[10] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 7–20. https://doi.org/10.1007/11554844_3

[11] Thorsten Berger, Marsha Chechik, Timo Kehrer, and Manuel Wimmer. 2019. Software Evolution in Time and Space: Unifying Version and Variability Management (Dagstuhl Seminar 19191). *Dagstuhl Reports* 9, 5 (2019), 1–30. https://doi.org/10.4230/DagRep.9.5.1

[12] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 16–25. https://doi.org/10.1145/2791060.2791108

[13] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. 1993. The Concept Assignment Problem in Program Understanding. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 482–498. https://doi.org/10.1109/ICSE.1993.346017

[14] Philip Bille. 2005. A Survey on Tree Edit Distance and Related Problems. *Theoretical Computer Science* 337, 1–3 (2005), 217–239. https://doi.org/10.1016/j.tcs.2004.12.030

[15] Paul Maximilian Bittner. 2020. *Semi-Automated Inference of Feature Traceability During Software Development*. Master's thesis. TU Braunschweig. https://doi.org/10.24355/dbbs.084-202002271120-0

[16] Goetz Botterweck and Andreas Pleuss. 2014. Evolution of Software Product Lines. In *Evolving Software Systems*. Springer, 265–295. https://doi.org/10.1007/978-3-642-45398-4_9

[17] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change Detection in Hierarchically Structured Information. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*. ACM, 493–504. https://doi.org/10.1145/233269.233366

[18] Reidar Conradi and Bernhard Westfechtel. 1998. Version Models for Software Configuration Management. *ACM Computing Surveys (CSUR)* 30, 2 (1998), 232–282. https://doi.org/10.1145/280277.280280

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

[20] Michiel de Wit, Andy Zaidman, and Arie van Deursen. 2009. Managing Code Clones Using Dynamic Change Tracking and Resolution. In *Proc. Int'l Conf. on Software Maintenance (ICSM)*. 169–178. https://doi.org/10.1109/ICSM.2009.5306336

[21] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2018. FEVER: An Approach to Analyze Feature-Oriented Changes and Artefact Co-Evolution in Highly Configurable Systems. *Empirical Software Engineering (EMSE)* 23, 2 (2018), 905–952. https://doi.org/10.1007/s10664-017-9557-6

[22] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature Location in Source Code: A Taxonomy and Survey. *J. Software: Evolution and Process* 25, 1 (2013), 53–95. https://doi.org/10.1002/smr.567

[23] Ekwa Duala-Ekoko and Martin P. Robillard. 2010. Clone Region Descriptors: Representing and Tracking Duplication in Source Code. *Trans. on Software Engineering and Methodology (TOSEM)* 20, 1 (2010), 31 pages. https://doi.org/10.1145/1767751.1767754

[24] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *Proc. Europ. Conf. on Software Maintenance and Reengineering (CSMR)*. IEEE, 25–34. https://doi.org/10.1109/CSMR.2013.13

[25] Andrew David Eisenberg and Kris De Volder. 2005. Dynamic Feature Traces: Finding Features in Unfamiliar Code. In *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE, 337–346. https://doi.org/10.1109/ICSM.2005.42

[26] Sina Entekhabi, Anton Solback, Jan-Philipp Steghöfer, and Thorsten Berger. 2019. Visualization of Feature Locations with the Tool FeatureDashboard. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 1–4. https://doi.org/10.1145/3307630.3342392

[27] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. 313–324. https://doi.org/10.1145/2642937.2642982

[28] Wolfram Fenske, Jens Meinicke, Sandro Schulze, Steffen Schulze, and Gunter Saake. 2017. Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line. In *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 316–326. https://doi.org/10.1109/SANER.2017.7884632

[29] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 665–668. https://doi.org/10.1109/ICSE.2015.218

[30] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. on Software Engineering (TSE)* 33, 11 (2007), 725–743. https://doi.org/10.1109/TSE.2007.70731

[31] Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *Proc. Working Conf. on Reverse Engineering (WCRE)*. 279–288. https://doi.org/10.1109/WCRE.2008.44

[32] Florian Heidenreich, Jan Kopcsek, and Christian Wende. 2008. FeatureMapper: Mapping Features to Models. In *Companion Int'l Conf. on Software Engineering (ICSEC)*. ACM, 943–944. https://doi.org/10.1145/1370175.1370199 Informal demonstration paper.

[33] Patrick Heymans, Quentin Boucher, Andreas Classen, Arnaud Bourdoux, and Laurent Demonceau. 2012. A Code Tagging Approach to Software Product Line Development. *Int'l J. Software Tools for Technology Transfer (STTT)* 14 (2012), 553–566. Issue 5. https://doi.org/10.1007/s10009-012-0242-1

[34] James W. Hunt and Thomas Gregory Szymanski. 1977. A Fast Algorithm for Computing Longest Common Subsequences. *Comm. ACM* 20, 5 (1977), 350–353. https://doi.org/10.1145/359581.359603

[35] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability with Embedded Annotations. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 61–70. https://doi.org/10.1145/2791060.2791107

[36] Huzefa Kagdi, Malcom Gethers, and Denys Poshyvanyk. 2013. Integrating Conceptual and Logical Couplings for Change Impact Analysis in Software. *Empirical Software Engineering (EMSE)* 18, 5 (2013), 933–969. https://doi.org/10.1007/s10664-012-9233-9

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

[38] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 311–320. https://doi.org/10.1145/1368088.1368131

[39] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don Batory. 2009. Guaranteeing Syntactic Correctness for All Product Line Variants: A Language-Independent Approach. In *Proc. Int'l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE)*, Manuel Oriol and Bertrand Meyer (Eds.). Springer, 175–194. https://doi.org/10.1007/978-3-642-02571-6_11

[40] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. 2014. Variability Mining: Consistent Semiautomatic Detection of Product-Line Features. *IEEE Trans. on Software Engineering (TSE)* 40, 1 (2014), 67–82. https://doi.org/10.1109/TSE.2013.45

[41] Timo Kehrer. 2015. *Calculation and Propagation of Model Changes Based on User-Level Edit Operations: A Foundation for Version and Variant Management in Model-Driven Engineering*. Ph.D. Dissertation. University of Siegen.

[42] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 2011. A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 163.–172. https://doi.org/10.1109/ASE.2011.6100050

[43] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 2013. Consistency-Preserving Edit Scripts in Model Versioning. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 191–201. https://doi.org/10.1109/ASE.2013.6693079

[44] Timo Kehrer, Thomas Thüm, Alexander Schultheiß, and Paul Maximilian Bittner. 2021. Bridging the Gap Between Clone-and-Own and Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 21–25. https://doi.org/10.1109/ICSE-NIER52604.2021.00013

[45] Benjamin Klatt, Martin Küster, and Klaus Krogmann. 2013. A Graph-Based Analysis Concept to Derive a Variation Point Design from Product Copies. In *Proc. Int'l Workshop on Reverse Variability Engineering (REVE)*. 1–8.

[46] Rainer Koschke. 2007. Survey of Research on Software Clones. In *Duplication, Redundancy, and Similarity in Software (Dagstuhl Seminar Proceedings, 06301)*, Rainer Koschke, Ettore Merlo, and Andrew Walenstein (Eds.). IBFI. http://drops.dagstuhl.de/opus/volltexte/2007/962

[47] Rainer Koschke, Pierre Frenzel, Andreas P. Breu, and Karsten Angstmann. 2009. Extending the Reflexion Method for Consolidating Software Variants into Product Lines. *Software Quality Journal (SQJ)* 17, 4 (2009), 331–366. https://doi.org/10.1007/s11219-009-9077-8

[48] Jacob Krüger, Thorsten Berger, and Thomas Leich. 2019. Features and How to Find Them: A Survey of Manual Feature Location. In *Software Engineering for*

Variability Intensive Systems - Foundations and Applications. Auerbach Publications / Taylor & Francis, 153–172. https://doi.org/10.1201/9780429022067-7

[49] Bruno Lague, Daniel Proulx, Jean Mayrand, Ettore M. Merlo, and John Hudepohl. 1997. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE, 314–321. https://doi.org/10.1109/ICSM.1997.624264

[50] Miguel A. Laguna and Yania Crespo. 2013. A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring. *Science of Computer Programming (SCP)* 78, 8 (2013), 1010–1034. https://doi.org/10.1016/j.scico.2012.05.003

[51] Raúl Lapeña, Manuel Ballarin, and Carlos Cetina. 2016. Towards Clone-and-Own Support: Locating Relevant Methods in Legacy Products. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 194–203. https://doi.org/10.1145/2934466.2934485

[52] Duc Le, Eric Walkingshaw, and Martin Erwig. 2011. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. In *Proc. Int'l Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 143–150. https://doi.org/10.1109/VLHCC.2011.6070391

[53] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 105–114. https://doi.org/10.1145/1806799.1806819

[54] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proc. Int'l Conf. on Aspect-Oriented Software Development (AOSD)*. ACM, 191–202. https://doi.org/10.1145/1960275.1960299

[55] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 49–62. https://doi.org/10.1145/3136040.3136054

[56] Lukas Linsbauer, Alexander Egyed, and Roberto Erick Lopez-Herrejon. 2016. A Variability Aware Configuration Management and Revision Control Platform. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 803–806. https://doi.org/10.1145/2889160.2889262

[57] Lukas Linsbauer, Stefan Fischer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. Using Traceability for Incremental Construction and Evolution of Software Product Portfolios. In *Proc. Int'l Symposium on Software and Systems Traceability (SST)*. IEEE, 57–60. https://doi.org/10.1109/SST.2015.16

[58] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability Extraction and Modeling for Product Variants. *Software and System Modeling (SoSyM)* 16, 4 (2017), 1179–1199. https://doi.org/10.1007/s10270-015-0512-y

[59] Jia Liu, Don Batory, and Christian Lengauer. 2006. Feature Oriented Refactoring of Legacy Applications. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 112–121. https://doi.org/10.1145/1134285.1134303

[60] Wardah Mahmood, Daniel Strueber, Thorsten Berger, Ralf Laemmel, and Mukelabai Mukelabai. 2021. Seamless Variability Management With the Virtual Platform. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 1658–1670. https://doi.org/10.1109/ICSE43902.2021.00147

[61] R. Malaquias, M. Ribeiro, R. Bonifácio, E. Monteiro, F. Medeiros, A. Garcia, and R. Gheyi. 2017. The Discipline of Preprocessor-Based Annotations - Does #ifdef TAG n't #endif Matter. In *Proc. Int'l Conf. on Program Comprehension (ICPC)*. 297–307. https://doi.org/10.1109/ICPC.2017.41

[62] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Bottom-Up Adoption of Software Product Lines: A Generic and Extensible Approach. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 101–110. https://doi.org/10.1145/2791060.2791086

[63] Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. 2013. Investigating Preprocessor-Based Syntax Errors. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 75–84. https://doi.org/10.1145/2517208.2517221

[64] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. 2018. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Trans. on Software Engineering (TSE)* 44, 5 (2018), 453–469. https://doi.org/10.1109/TSE.2017.2688333

[65] Raul Medina-Mora and Peter H. Feiler. 1981. An Incremental Programming Environment. *IEEE Trans. on Software Engineering (TSE)* 5 (1981), 472–482.

[66] Thilo Mende, Rainer Koschke, and Felix Beckwermert. 2009. An Evaluation of Code Similarity Identification for the Grow-and-Prune Model. *J. Software Maintenance and Evolution (JSME)* 21, 2 (2009), 143–169. https://doi.org/10.1002/smr.v21:2

[67] Gabriela Karoline Michelon, Lukas Linsbauer, Wesley K.G. Assunção, Stefan Fischer, and Alexander Egyed. 2021. A Hybrid Feature Location Technique for Re-Engineering Single Systems into Software Product Lines. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 11, 9 pages. https://doi.org/10.1145/3442391.3442403

[68] Gabriela Karoline Michelon, David Obermann, Wesley Klewerton Guez Assunção, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. 2020. Mining Feature Revisions in Highly-Configurable Software Systems. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 74–78. https://doi.org/10.1145/3382026.3425776

[69] Gabriela Karoline Michelon, David Obermann, Lukas Linsbauer, Wesley Klewerton Guez Assunção, Paul Grünbacher, and Alexander Egyed. 2020. Locating Feature Revisions in Software Systems Evolving in Space and Time. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, Article 14, 11 pages. https://doi.org/10.1145/3382025.3414954

[70] Webb Miller and Eugene W. Myers. 1985. A File Comparison Program. *Software: Practice and Experience* 15, 11 (1985), 1025–1040. https://doi.org/10.1002/spe.4380151102

[71] Eugene W. Myers. 1986. An O(ND) Difference Algorithm and Its Variations. *Algorithmica* 1, 1-4 (1986), 251–266. https://doi.org/10.1007/BF01840446

[72] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. 2012. Clone Management for Evolving Software. *IEEE Trans. on Software Engineering (TSE)* 38, 5 (2012), 1008–1026. https://doi.org/10.1109/TSE.2011.90

[73] David Notkin. 1985. The GANDALF Project. *J. Systems and Software (JSS)* 5, 2 (1985), 91–105. https://doi.org/10.1016/0164-1212(85)90011-1

[74] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wąsowski, and Paulo Borba. 2013. Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 91–100. https://doi.org/10.1145/2491627.2491628

[75] Mateusz Pawlik and Nikolaus Augsten. 2011. RTED: A Robust Algorithm for the Tree Edit Distance. *Computing Research Repository (CoRR)* 5, 4 (2011), 334–345.

[76] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. 2016. Synchronizing Software Variants with VariantSync. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 329–332. https://doi.org/10.1145/2934466.2962726

[77] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer. https://doi.org/10.1007/3-540-28901-1

[78] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. 2007. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Trans. on Software Engineering (TSE)* 33, 6 (2007), 420–432. https://doi.org/10.1109/TSE.2007.1016

[79] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software Clone Detection: A Systematic Review. *J. Information and Software Technology (IST)* 55, 7 (2013), 1165–1199. https://doi.org/10.1016/j.infsof.2013.01.008

[80] Thomas Reps and Tim Teitelbaum. 1984. The Synthesizer Generator. *SIGPLAN Not.* 19, 5 (1984), 42–48. https://doi.org/10.1145/800020.808247

[81] Nicholas Rescher. 1968. *Many-Valued Logic*. Springer, 54–125. https://doi.org/10.1007/978-94-017-3546-9_6

[82] Matthias Rieger, Stéphane Ducasse, and Georges Golomingi. 1999. Tool Support for Refactoring Duplicated OO Code. In *Proc. Europ. Conf. on Object-Oriented Programming (ECOOP)*, Vol. 1743. Springer, 177–178.

[83] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering: Product Lines, Languages, and Conceptual Models*, Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and Jorn Bettin (Eds.). Springer, 29–58. https://doi.org/10.1007/978-3-642-36654-3_2

[84] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2013. Managing Cloned Variants: A Framework and Experience. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 101–110. https://doi.org/10.1145/2491627.2491644

[85] Thomas Schmorleiz and Ralf Lämmel. 2014. Similarity Management via History Annotation. In *Proc. Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE)*. Dipartimento di Informatica Università degli Studi dell'Aquila, L'Aquila, Italy, 45–48.

[86] Thomas Schmorleiz and Ralf Lämmel. 2016. Similarity Management of 'Cloned and Owned' Variants. In *Proc. ACM Symposium on Applied Computing (SAC)*. ACM, 1466–1471. https://doi.org/10.1145/2851613.2851785

[87] Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. 2013. Does the Discipline of Preprocessor Annotations Matter?: A Controlled Experiment. *SIGPLAN Not.* 49, 3 (2013), 65–74. https://doi.org/10.1145/2637365.2517215

[88] Felix Schwägerl and Bernhard Westfechtel. 2016. SuperMod: Tool Support for Collaborative Filtered Model-Driven Software Product Line Engineering. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 822–827. https://doi.org/10.1145/2970276.2970288

[89] Janet Siegmund. 2016. Program Comprehension: Past, Present, and Future. In *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 13–20. https://doi.org/10.1109/SANER.2016.35

[90] Bolesław Sobociński. 1952. Axiomatization of a Partial System of Three-Value Calculus of Propositions. *Journal of Computing Systems* 1, 1 (1952), 23–55. https://doi.org/10.2307/2267445

[91] Henry Spencer and Geoff Collyer. 1992. #ifdef Considered Harmful, or Portability Experience With C News. In *USENIX*. USENIX Association, 185–197.
[92] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. 2016. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 323–333. https://doi.org/10.1109/ICSME.2016.88
[93] Stefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 151–160. https://doi.org/10.1109/ICSM.2015.7332461
[94] Matús Sulír, Milan Nosál, and Jaroslav Porubän. 2018. Recording Concerns in Source Code Using Annotations. *Computing Research Repository (CoRR)* abs/1808.03576 (2018).
[95] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Towards Efficient Analysis of Variation in Time and Space. In *Proc. Int'l Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution)*. ACM, 57–64. https://doi.org/10.1145/3307630.3342414
[96] Michael Toomim, Andrew Begel, and Susan L. Graham. 2004. Managing Duplicated Code with Linked Editing. In *Proc. Int'l Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 173–180. https://doi.org/10.1109/VLHCC.2004.35
[97] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *Proc. Int'l Conf. on Software Language Engineering (SLE)*. Springer, 41–61. https://doi.org/10.1007/978-3-319-11245-9_3
[98] Anneliese von Mayrhauser, A. Marie Vans, and Adele E. Howe. 1997. Program Understanding Behaviour During Enhancement of Large-Scale Software. *J. Software: Evolution and Process* 9, 5 (1997), 299–327. https://doi.org/10.1002/(SICI)1096-908X(199709/10)9:5<299::AID-SMR157>3.0.CO;2-S
[99] Neil Walkinshaw, Marc Roper, and Murray Wood. 2007. Feature Location and Extraction using Landmarks and Barriers. In *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE, 54–63. https://doi.org/10.1109/ICSM.2007.4362618
[100] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2013. How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study. *J. Software: Evolution and Process* 25, 11 (2013), 1193–1224. https://doi.org/10.1002/smr.1593 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1593
[101] Norman Wilde and Michael C. Scully. 1995. Software Reconnaissance: Mapping Program Features to Code. *J. Software Maintenance: Research and Practice* 7, 1 (1995), 49–62. https://doi.org/10.1002/smr.4360070105
[102] David Wille, Sandro Schulze, Christoph Seidl, and Ina Schaefer. 2016. Custom-Tailored Variability Mining for Block-Based Languages. In *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 271–282. https://doi.org/10.1109/SANER.2016.13
[103] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. 2012. Feature Location in a Collection of Product Variants. In *Proc. Working Conf. on Reverse Engineering (WCRE)*. IEEE, 145–154. https://doi.org/10.1109/WCRE.2012.24
[104] Tewfik Ziadi, Christopher Henard, Mike Papadakis, Mikal Ziane, and Yves Le Traon. 2014. Towards a Language-Independent Approach for Reverse-Engineering of Software Product Lines. In *Proc. ACM Symposium on Applied Computing (SAC)*. ACM, 1064–1071. https://doi.org/10.1145/2554850.2554874