# Foundations of Collaborative, Real-Time Feature Modeling

Elias Kuiter
Otto-von-Guericke-University
Magdeburg, Germany
kuiter@ovgu.de

Sebastian Krieter
Harz University of Applied Sciences
Otto-von-Guericke-University
Wernigerode & Magdeburg, Germany
skrieter@hs-harz.de

Jacob Krüger
Otto-von-Guericke-University
Magdeburg, Germany
jkrueger@ovgu.de

Thomas Leich
Harz University of Applied Sciences
Wernigerode, Germany
tleich@hs-harz.de

Gunter Saake
Otto-von-Guericke-University
Magdeburg, Germany
saake@ovgu.de

## ABSTRACT

Feature models are core artifacts in software-product-line engineering to manage, maintain, and configure variability. Feature modeling can be a cross-cutting concern that integrates technical and business aspects of a software system. Consequently, for large systems, a team of developers and other stakeholders may be involved in the modeling process. In such scenarios, it can be useful to utilize collaborative, real-time feature modeling, analogous to collaborative text editing in Google Docs or Overleaf. However, current techniques and tools only support a single developer to work on a model at a time. Collaborative and simultaneous editing of the same model is often achieved by using version control systems, which can cause merge conflicts and do not allow immediate verification of a model, hampering real-time collaboration outside of face-to-face meetings. In this paper, we describe the formal foundations of collaborative, real-time feature modeling, focusing on concurrency control by synchronizing multiple actions of collaborators in a distributed network. We further report on preliminary results, including an initial prototype. Our contribution provides the basis for extending feature-modeling tools to enable advanced collaborative feature modeling and integrate it with related tasks.

## CCS CONCEPTS

• **Software and its engineering → Feature interaction**; **Software product lines**; **Programming teams**.

## KEYWORDS

Software Product Line, Groupware, Feature Modeling, Consistency Maintenance, Collaboration

**ACM Reference Format:**
Elias Kuiter, Sebastian Krieter, Jacob Krüger, Thomas Leich, and Gunter Saake. 2019. Foundations of Collaborative, Real-Time Feature Modeling. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3336294.3336308

## 1 INTRODUCTION

Variability modeling is a core activity for developing and managing a Software Product Line (SPL) [3, 19, 42]. It does not only concern implementation artifacts, but various other aspects of an SPL as well [7, 18]. For instance, a variability model often incorporates design decisions specific to an SPL's domain, but also provides a layer of abstraction that end users can comprehend. Thus, for large-scale projects, multiple people must work corporately to create a meaningful variability model [6, 38].

Numerous tools facilitate variability modeling with specialized user interfaces and automated model analyses (e.g., *FeatureIDE* [35], *pure::variants* [8], and *Gears* [28]). However, we are not aware of a technique for variability modeling that supports *collaborative, real-time editing*, similar to Google Docs or Overleaf, which hampers efficient cooperation during the modeling process. To the best of our knowledge, existing tools allow only a single user to edit a variability model at a time. Using version control systems, such as Git, developers can share and distribute variability models, but this is neither in real-time nor does it support a semantically meaningful resolution of conflicts. We see various potential use cases for collaborative, real-time variability modeling, for example:

- Multiple domain engineers can work simultaneously on the same variability model, either on different tasks (e.g., editing existing constraints) or on a coordinated task (e.g., introducing a set of new features).
- Engineers can share and discuss the variability model with domain experts, allowing to evolve it with real-time feedback without requiring costly co-location of the participants.
- Lecturers can teach variability-modeling concepts in a collaborative manner and can more easily involve the audience in hands-on exercises.

The most established form of variability models are *feature models* and their visual representations, *feature diagrams* [7, 16, 19]—on which we focus in this paper. Feature models capture the features of an SPL and their interdependencies, thereby defining the user-visible functionalities that are common or different for variants.

In this paper, we describe the conceptual foundations for our technique to achieve collaborative, real-time feature modeling. More precisely, we define the requirements our technique needs to fulfill

---

*SPLC '19, September 9–13, 2019, Paris, France*

ACM ISBN 978-1-4503-7138-4/19/09…$15.00
https://doi.org/10.1145/3336294.3336308

and, based on these requirements, derive a formal description of collaborative, real-time feature modeling that allows us to ensure its correct behavior and guides the actual implementation. This includes defining a basic set of feature-modeling operations, a conflict relation between these operations, and mechanisms for detecting and resolving potential conflicts. For this purpose, we extend existing techniques for real-time collaboration to provide the basis for our and future collaborative, real-time variability modeling techniques, which may be integrated into existing tools. Although we have implemented an initial prototype to demonstrate the feasibility of our technique, we do not elaborate on its technical details. Overall, we make the following contributions:

- We identify and describe what requirements should be fulfilled by a collaborative, real-time feature modeling technique and a corresponding editor.
- We define a concurrency control technique to allow for collaborative, real-time feature modeling. In particular, we introduce strategies and mechanisms to detect and resolve conflicts, thereby ensuring that the edited feature models remain syntactically and semantically consistent.
- We briefly report on preliminary results of our technique with regard to formal correctness and an initial prototype.

We aim to support uses cases that are based on three general conditions. First, we assume that users need or want to work simultaneously on the same feature model, for instance, to coordinate their efforts when performing independent or interacting tasks [33]. Thus, mechanisms for concurrency control are required. Second, we assume that a rather small team (i.e., no more than ten developers) is maintaining a feature model, based on studies on real-world SPLs [6, 24, 38]. For larger teams, managing collaborations and automatically resolving conflicts becomes much harder. Finally, we assume that not all developers work co-located, but are remotely connected [33], for which we aim to support both peer-to-peer and client-server architectures [44].

## 2 FORMAL FOUNDATIONS

Within this paper, we present a formal technique for collaborative, real-time feature modeling. In the following, we briefly introduce the notation of feature models and key concepts regarding consistency maintenance in collaborative editing systems.

### 2.1 Feature Modeling

We consider feature models in the form of feature diagrams, which specify the variability of SPLs using a hierarchy of features and cross-tree constraints. Thus, we define a feature model as follows:

**Definition 1.** *A* feature model *FM is a tuple* $(\mathcal{F}, \mathcal{C})$ *where* $\mathcal{F}$ *is a finite set of features and* $\mathcal{C}$ *is a finite set of cross-tree constraints.*[1]

*A* feature $F \in FM.\mathcal{F}$ *is a tuple (ID, parentID, optional, groupType, abstract, name) where* $ID \in \mathcal{ID}$, $parentID \in \mathcal{ID} \cup \{\bot, \dagger\}$, $optional \in \{true, false\}$, $groupType \in \{and, or, alternative\}$, $abstract \in \{true, false\}$, *and name is a string.*

*A* cross-tree constraint $C \in FM.\mathcal{C}$ *is a tuple* $(ID, \phi)$ *where* $ID \in \mathcal{ID}$ *and* $\phi$ *is a propositional formula with variables ranging over* $\mathcal{ID}$, *i.e.,* $Var(\phi) \subseteq \mathcal{ID}$.

*The set* $\mathcal{ID}$ *contains all Universally Unique Identifiers (UUIDs) that can be used within a feature model.* $\bot$ *denotes the parentID of the root feature, while* $\dagger$ *denotes the parentID of an uninitialized feature.*

To simplify our definitions, we declare the two sets $\mathcal{F}_{FM}^{ID}$ and $\mathcal{C}_{FM}^{ID}$ for feature and constraint identifiers and the parent-child relation between features *descends from* ($\leq_{FM}$) as follows:

**Definition 2.** *Let FM be a feature model. Further, define*

- $\mathcal{F}_{FM}^{ID} := \{F.ID \mid F \in FM.\mathcal{F}\}$,
- $\mathcal{C}_{FM}^{ID} := \{C.ID \mid C \in FM.\mathcal{C}\}$, *and*
- $\leq_{FM}$ *as the reflexive transitive closure of* $\{(A.ID, B^{ID}) \mid A \in FM.\mathcal{F}, B^{ID} \in \mathcal{F}_{FM}^{ID} \cup \{\bot, \dagger\} \wedge A.parentID = B^{ID}\}$.

Using our notation from Definition 1, we can formally describe any feature model—using its feature diagram representation. However, this definition still allows that integral properties of feature models are violated. These properties are important, as we intend to manipulate feature models by means of operations. Thus, we also define the following conditions to describe *legal* feature models:

**Definition 3.** *A feature model FM is considered* legal *iff all of the following conditions are true:*

- Unique identifiers: $|FM.\mathcal{F}| = |\mathcal{F}_{FM}^{ID}| \wedge |FM.\mathcal{C}| = |\mathcal{C}_{FM}^{ID}|$
- Valid parents: $\forall F \in FM.\mathcal{F} : F.parentID \in \mathcal{F}_{FM}^{ID} \cup \{\bot, \dagger\}$
- Valid constraints: $\forall C \in FM.\mathcal{C} : Var(C.\phi) \subseteq \mathcal{F}_{FM}^{ID}$
- Single root: $\exists! F \in FM.\mathcal{F} : F.parentID = \bot$
- Acyclic: $\forall F^{ID} \in \mathcal{F}_{FM}^{ID} : F^{ID} \leq_{FM} \bot \vee F^{ID} \leq_{FM} \dagger$

*We denote the set of all legal feature models as* $\mathcal{FM}$.

We utilize this formalization of feature models for defining our operation model in Section 3.

### 2.2 Consistency Maintenance

Real-time, remote collaborative editing systems, usually rely on *operations* to propagate changes among connected users [43]. An operation is the description of an atomic manipulation of a document with a distinct intention. It is applied to a document to transform it from an old to a new (modified) state. We adopt this operational concept and use the definitions of Sun et al. [53] to formally describe *concurrency* and *conflict*.

*Concurrency.* Multiple users can create operations at different sites at different times. However, the synchronization of these operations between sites is affected by network latency, and thus not instant. Consequently, the order of submitted operations cannot be simply tracked based on physical time. Instead, we adapt a well-known strict partial order [30, 34, 53] to determine the temporal (and thus causal) relationships of operations and define the notion of concurrency.

**Definition 4** (Causally-Preceding Relation [53])**.** *Let* $O_a$ *and* $O_b$ *be two operations generated at sites i and j, respectively. Then,* $O_a \rightarrow O_b$ *(*$O_a$ causally precedes $O_b$*) iff at least one of the following is true:*

- $i = j$ *and* $O_a$ *is generated* before $O_b$

---

[1] For easier readability, we use a dot notation to access a tuple's elements (or *attributes*). For instance, $FM.\mathcal{F}$ refers to the features in the feature model *FM*. Further, we interpret $\mathcal{F}$ and $\mathcal{C}$ as functions to facilitate attribute lookup; e.g., $FM.\mathcal{F}(F^{ID})$ refers to the feature (uniquely) identified by $F^{ID}$ in *FM*.

- *$i \neq j$ and at site $j$, $O_a$ is executed* before *$O_b$ is generated*
- $\exists O_x : O_a \rightarrow O_x \wedge O_x \rightarrow O_b$

*where* before *refers to a site's local physical time. Further, $O_a$ and $O_b$ are said to be* concurrent *iff $O_a \not\rightarrow O_b$ and $O_b \not\rightarrow O_a$.*

*Conflict.* Several challenges hamper the maintenance of a consistent document state in collaborative, real-time editors [51, 53]. Of particular interest is the *intention violation* problem, which is concerned with conflicts. A conflict occurs if two or more concurrent operations violate each other's intentions. For example, two operations that set the name of the same feature to different values are intention-violating (i.e., in conflict), as both override the other operation's effect. In Section 5, we describe how this problem may be solved in the context of feature modeling.

## 3 OPERATION MODEL

A collaborative feature model editor must support a variety of operations to achieve a similar user experience as single-user editors. However, supporting various operations can lead to more interactions between operations, which makes consistency checking and resolving of conflicts more complex. Furthermore, it hampers reasoning about the editor's correctness. To address this issue, we use a two-layered operation architecture [54], in which we separate two kinds of operations: low-level Primitive Operations (POs) and high-level Compound Operations (COs). POs represent fine-grained edits to feature models and are suitable to use in concurrency control techniques, as they are simple and composable. A CO is an ordered sequence of POs and exposes an actual feature-modeling operation to the application.

Using this two-layer architecture, instead of one large set of operations, has several advantages: When detecting conflicts between operations, we can focus on POs and do not need to analyze any high-level COs, as they are PO sequences. Also, to extend an editor with additional operations, we only need to implement new COs, without making major changes to the conflict detection. In the following, we exemplify POs and COs.

*Primitive Operations.* Single-user feature modeling tools allow creating, removing, and modifying features and cross-tree constraints in various ways. We present two exemplary POs that serve as building blocks for such COs. For each PO, we provide formal semantics in the form of pre- and postconditions, where *FM* and *FM′* refer to the feature model before and after applying the PO, respectively. By convention, no PO shall have any other side effects than those specified in the postconditions.

**createFeaturePO**($F^{ID}$): Creates a feature with a globally unique identifier and default attributes, not yet inserted to the feature tree.

PRE: $F^{ID} \in \mathcal{ID}$
$F^{ID} \notin \mathcal{F}^{ID}_{FM}$

POST: $(F^{ID}, \dagger, \mathit{true}, \mathit{and}, \mathit{false}, \texttt{NewFeature}) \in FM'.\mathcal{F}$

**updateFeaturePO**($F^{ID}$, *attr*, *oldVal*, *newVal*): Updates an attribute *attr* of a feature $F^{ID}$ to a new value *newVal*. The old attribute value is included as well to facilitate conflict detection. *Dom* refers to an attribute's domain (cf. Definition 1). Further, $FM.\mathcal{F}(F^{ID}).[\mathit{attr}]$ refers to a particular feature attribute value as specified by *attr*.

PRE: $F^{ID} \in \mathcal{F}^{ID}_{FM}$
$\mathit{attr} \in \{\mathit{parentID}, \mathit{optional}, \mathit{groupType}, \mathit{abstract}, \mathit{name}\}$
$\mathit{oldVal} = FM.\mathcal{F}(F^{ID}).[\mathit{attr}]$
$\mathit{newVal} \in \mathit{Dom}(\mathit{attr})$

POST: $FM'.\mathcal{F}(F^{ID}).[\mathit{attr}] = \mathit{newVal}$

For cross-tree constraints, we define analogous POs.

*Compound Operations.* To allow for high-level modeling operations, we employ COs. Each CO consists of a sequence of atomically applied POs. Further, each CO has associated preconditions and an algorithm that generates the CO's PO sequence, which must fulfill the preconditions of each comprised PO. Whenever a user requests to execute a CO, we have to check the preconditions against the current feature model *FM*, and then invoke the CO's algorithm with *FM* and any required arguments (e.g., a feature parent *FP*). We then apply the generated CO locally and propagate it to other collaborators. For the sake of brevity, we only show one exemplary CO, which creates a feature below another feature:

```
function createFeatureBelow($FM$, $F^{ID}$, $FP^{ID}$)
  Require: $F^{ID} \in \mathcal{ID}$, $F^{ID} \notin \mathcal{F}^{ID}_{FM}$, $FP^{ID} \in \mathcal{F}^{ID}_{FM}$
  return [createFeaturePO($F^{ID}$),
          updateFeaturePO($F^{ID}$, parentID, $\dagger$, $FP^{ID}$)]
end function
```

We defined additional operations, such as (re)moving features and cross-tree constraints [29], and more can be designed in the future.

*Operation Application.* As POs and COs are only descriptions of manipulations on a feature model, we further need to define how to apply them to produce a new (modified) feature model.

**Definition 5.** *Let $FM \in \mathcal{FM}$. Further, let PO and CO be a primitive and a compound operation whose preconditions are satisfied with regard to FM. Then, $FM' = \mathit{applyPO}(FM, PO)$ denotes the feature model $FM'$ that results from applying PO to FM. Further, we define $\mathit{applyCO}(FM, CO)$ as the subsequent application of all primitive operations contained in CO to FM with applyPO.*

We assume the tool to provide *applyPO*, which ensures all postconditions of primitive operations. Note that *applyCO* does treat every compound operation equally, which facilitates conflict detection and future extensions. Further, we can already derive that *applyCO* always preserves the legality of feature models (cf. Definition 3) in a single-user scenario [29].

## 4 REQUIREMENTS ANALYSIS

Before developing a technique for collaborative, real-time feature modeling, we need to define the requirements that such a technique must fulfill according to the general conditions of our considered use cases. We then discuss a concurrency control technique for collaborative editing which fits our requirements.

### 4.1 Requirements

To allow users to work on the same feature model simultaneously, we define four requirements (Req) based on the general conditions we described in Section 1. This list is not complete, but focuses on formal requirements that enable our technique and its integration.

*$Req_1$: Concurrency.* The most important requirement for enabling collaborative, real-time feature modeling is that our technique must

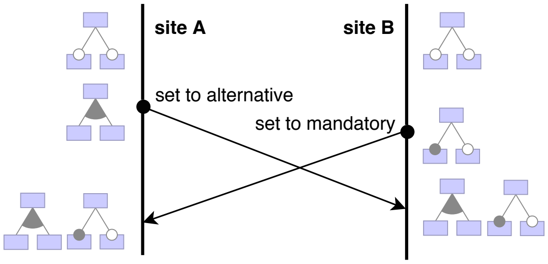

**Figure 1: The MVMD technique applied to feature modeling.**

allow multiple users to concurrently access and edit a model [22, 23, 26]. Consequently, our technique must incorporate a concurrency control technique to manage concurrent operations. Without such a technique, concurrency can lead to inconsistency and confusion.

*Req$_2$: Intention Preservation.* A crucial requirement for modeling and specification activities is that an editor accurately reflects an operation's expected behavior [13, 26, 51, 53]. This means that collaborators submit an operation and expect the system to apply and retain the intended change. To this end, our technique must ensure that the result is consistent in itself, but also to the issued operation with respect to multiple, potentially conflicting operations that are issued by various collaborators. For our technique, we require a method that prevents unexpected results, such as masked, overridden, and inconsistent operations.

*Req$_3$: Optimism.* A collaborative, real-time editor may process operations using a *pessimistic* or *optimistic* strategy [26, 43, 53]. Pessimistic strategies require all sites to acknowledge a change before it is carried out. Thus, such strategies include additional transmissions, which block the local user interface for that time. In contrast, optimistic strategies immediately apply changes locally, then propagate them to all other sites [9, 22, 26, 43]; relying on the users to resolve all occurring conflicts afterwards—assuming that conflicts occur only sparsely [13, 20, 23, 49]. With an optimistic strategy, the local system is always responsive and allows unconstrained collaboration as long as no conflict emerges. As we aim for a small team of remotely connected collaborators, we assume that during the editing process there is noticeable network latency, but only few conflicts. Thus, an optimistic strategy seems more suited for our technique, as it most likely improves editing experience compared to pessimistic strategies.

## 4.2 Multi-Version Multi-Display Technique

In the context of feature modeling, we argue that collaborators should be involved in resolving conflicts (similar to merging in version control systems) to preserve the intentions of all conflicting operations. To this end, we use a *multi-versioning concurrency control technique* [14, 49, 55]. In contrast to other techniques, multi-versioning techniques keep different versions of objects on which conflicting operations have been performed (similar to parent commits that are merged in version control systems). In particular, we focus on the Multi-Version Multi-Display (MVMD) technique [15, 50, 51], which lets users decide which new document version should be used in case of a conflict. In Figure 1, we show how this technique allocates two conflicting update operations on a feature model to two different versions, preserving intentions and allowing for subsequent manual conflict resolution. This technique encourages communication between collaborators and improves the confidence in the correctness of the resulting document. As MVMD fulfills all of our requirements, we use it as basis for our collaborative, real-time feature modeling technique and adapt it where necessary.

At the center of this technique is an application-specific *conflict relation*, which is used to determine algorithmically whether two operations are in conflict.

**Definition 6** (Conflict Relation [51, 56])**.** *A conflict relation* $\otimes$ *is a binary, irreflexive, and symmetric relation that indicates whether two given operations are* in conflict. *If two operations* $O_a$ *and* $O_b$ *are not in conflict with each other, i.e.,* $O_a \not\otimes O_b$, *they are* compatible. *Only concurrent operations may conflict, that is, for any operations* $O_a$ *and* $O_b$, $O_a \nparallel O_b \Rightarrow O_a \not\otimes O_b$.

Utilizing such a conflict relation, MVMD groups operations in suitable versions according to whether they are conflicting or compatible, as we show in Figure 1. We omit the details of how those versions may be constructed algorithmically and refer to the original paper [51]. In the following, we focus on our adaptations for feature modeling, which includes introducing a conflict relation (Section 5) as well as a conflict resolution process (Section 6) suitable for collaborative, real-time feature modeling.

# 5 CONFLICT DETECTION

In this section, we describe how we extended the MVMD technique with conflict relations for feature modeling to allow for the detection of conflicting operations. To this end, we briefly motivate and describe several required strategies and mechanisms.

## 5.1 Causal Directed Acyclic Graph

In Section 2.2, we introduced a causal ordering for tracking operations' concurrency relationships in the system. However, the *outer* and *inner conflict relations* for collaborative feature modeling (introduced below) require further information about causality relationships. To this end, we utilize that the causally-preceding relation is a strict partial order, and thus corresponds to a directed acyclic graph [34, 47]. Using such a Causal Directed Acyclic Graph (CDAG), we define the sets of Causally Preceding (CP) and Causally Immediately Preceding (CIP) operations for a given operation as follows:

**Definition 7.** *Let GO be a group of operations. The* causal directed acyclic graph *for GO is the graph* $G = (V, E)$ *where* $V = GO$ *is the set of vertices and* $E = \{(O_a, O_b) \mid O_a, O_b \in GO \land O_a \to O_b\}$ *is the set of edges. Then, the set of* causally preceding operations *for an* $O \in GO$ *is defined as* $CP_G(O) := \{O_a \mid (O_a, O) \in E\}$.
*Now, let* $(V, E')$ *be the transitive reduction of* $(V, E)$. *Then, the set of* causally immediately preceding operations *for an* $O \in GO$ *is defined as* $CIP_G(O) := \{O_a \mid (O_a, O) \in E'\}$.

The transitive reduction of a graph removes all edges that only represent transitive dependencies [2]. Therefore, an operation $O_a$ causally *immediately* precedes another operation $O_b$ when there is no operation $O_x$ such that $O_a \to O_x \to O_b$. Each collaborating site has a copy of the current CDAG, which is incrementally constructed and includes all previously generated and received operations.

## 5.2 Outer Conflict Relation

In its original context, the MVMD technique solely uses the operations' metadata to determine conflicts. However, no complex syntactic or semantic conflicts can be detected this way, because the underlying document is not available for conflict detection. In contrast, we propose that a conflict relation for feature modeling should not only consider an operation's metadata, but also the feature model. Such a conflict relation may inspect the involved operations and apply them to a suitable feature model to check whether their application introduces any inconsistencies.

In order to guarantee that such a suitable feature model exists for two given operations, all of their causally preceding operations must be compatible. Otherwise, the intention preservation property may be violated, so that the conflict relation would rely on potentially inconsistent and unexpected feature models.

The *outer conflict relation* (termed $\otimes_O$) serves to guarantee this property. It may be computed with outerConflicting, a recursive algorithm that uses the CDAG to propagate detected conflicts to all causally succeeding operations:

```
function outerConflicting($G$, $CO_a$, $CO_b$)
Require: $G$ is the CDAG for a group of operations $GO$,
         $CO_a, CO_b \in GO$
  if $CO_a \nparallel CO_b \lor CO_a = CO_b$ then return false
  if $\exists CIPO_a \in CIP_G(CO_a), CIPO_b \in CIP_G(CO_b)$:
      outerConflicting($G$, $CIPO_a$, $CIPO_b$)
    $\lor \exists CIPO_b \in CIP_G(CO_b)$: outerConflicting($G$, $CO_a$, $CIPO_b$)
    $\lor \exists CIPO_a \in CIP_G(CO_a)$: outerConflicting($G$, $CIPO_a$, $CO_b$)
  then return true
  return $CO_a \otimes_I CO_b$
end function
```

In the basic case, outerConflicting defers the conflict detection to the inner conflict relation ($\otimes_I$). The other cases ensure that there is a well-defined feature model for subsequent conflict detection, which enables us to check arbitrary consistency properties; with the disadvantage that few operations may falsely be flagged as conflicting. Using outerConflicting, we can compute $\otimes_O$ as follows:

**Definition 8.** *Two compound operations $CO_a$ and $CO_b$ are in* outer conflict, *i.e., $CO_a \otimes_O CO_b$, iff* outerConflicting*($G$, $CO_a$, $CO_b$) = true, where $G$ is the current CDAG at the site that executes* outerConflicting.

## 5.3 Topological Sorting Strategy

The outer conflict relation $\otimes_O$ ensures the existence of a suitable feature model. To actually produce such a feature model, we use

$$applyCOs(G, FM, COs) := reduce(applyCO, FM, topsort(COs, G))$$

where *topsort* corresponds to a topological sorting of operations according to their causality relationships specified in $G$, which *reduce* then applies one by one in that order to *FM*. We use applyCOs to apply (unordered) sets of mutually compatible operations to a feature model. Because the application order of operations is important for producing a correct result, our topological sorting strategy ensures that all causal relationships captured in the CDAG are respected.

## 5.4 Inner Conflict Relation

The *inner conflict relation* $\otimes_I$ detects conflicts that are specific to feature modeling. We introduce innerConflicting to determine $\otimes_I$ for two given compound operations:

```
function innerConflicting($G$, $FM$, $CO_a$, $CO_b$)
Require: $G$ is the CDAG for a group of operations $GO$,
         $FM$ is the initial feature model for $G$, $CO_a, CO_b \in GO$
  if $CO_a \nparallel CO_b \lor CO_a = CO_b$ then return false
  if syntacticallyConflicting($G, FM, CO_a, CO_b$)
  $\lor$ syntacticallyConflicting($G, FM, CO_b, CO_a$)
  then return true
  $FM \leftarrow$ applyCOs($G, FM, CP_G(CO_a) \cup CP_G(CO_b) \cup \{CO_a, CO_b\}$)
  return $\exists SP \in \mathcal{SP}$: SP($FM$) = true
end function
```

This algorithm makes use of syntacticallyConflicting, which determines whether two COs have a *syntactic conflict* that concerns basic syntactic properties of feature models. syntacticallyConflicting does so by applying both operations to a suitable feature model derived with applyCOs from the initial feature model. The second operation is applied step-wise, so we can inspect the current feature model for potential consistency problems using a set of *conflict detection rules* specific to feature modeling. These rules preserve the legality of feature models (cf. Definition 3) by detecting several problems, such as cycle-introducing operations and more [29].

To ensure the symmetry of $\otimes_I$, as required by Definition 6, innerConflicting uses syntacticallyConflicting to check for syntactic conflicts in both directions. Finally, innerConflicting may check additional arbitrary *semantic properties* on a feature model that includes the effects of $CO_a$ and $CO_b$. A semantic property $SP \in \mathcal{SP}$ is a deterministic function $SP\colon \mathcal{FM} \rightarrow \{true, false\}$ that returns whether a given legal feature model includes a semantic inconsistency. For instance, collaborators may want to ensure that the modeled SPL always has at least one product and does not include *dead*, *false-optional features* or any *redundant cross-tree constraints* [3, 5]. Note that the MVMD technique allows only pairwise conflict detection of operations, as interactions of higher order are hard to detect [4, 11, 12]. Using innerConflicting, we can compute $\otimes_I$ as follows:

**Definition 9.** *Two compound operations $CO_a$ and $CO_b$ are in* inner conflict, *i.e., $CO_a \otimes_I CO_b$, iff* innerConflicting*($G$, $FM$, $CO_a$, $CO_b$) = true, where $G$ and $FM$ are the current CDAG and initial feature model at the site that executes* innerConflicting.

Our conflict detection technique can now be implemented by using $\otimes_O$ as conflict relation in the MVMD technique [29, 51].

# 6 CONFLICT RESOLUTION

Our extension of the MVMD technique fully automates the detection of conflicts and allocation of feature-model versions. However, MVMD does not offer functionality for actually resolving conflicts. Thus, we propose a *manual conflict resolution process* (cf. Figure 2) during which collaborators examine alternative feature model versions and negotiate a specific version [49, 55]. To this end, we allow collaborators to cast *votes* for their preferred feature model versions, which allows for fair and flexible conflict resolution [21, 25, 37].

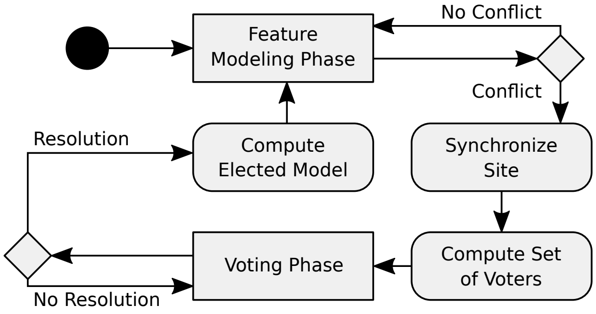

**Figure 2: Conflict resolution process.**

In our process, a site forbids any further editing (i.e., freeze site) when a conflict is detected. This forces collaborators to address the conflict, avoiding any further divergence. The freeze also ensures the correctness of our technique, as the MVMD technique has only been proven correct for this use case [51, 56]. After freezing, the system synchronizes all sites so that all collaborators are aware of all versions before starting the voting process, which is the only synchronization period our technique needs. Next, each site may flexibly compute a set of voters (i.e., collaborators that are eligible to vote) based on the collaborators' preferences. For example, a subset could contain only collaborators involved in the conflict or those with elevated rights. To start the voting, we initialize a set of vote results as an empty set. In the voting phase, every voter may cast a vote on a single feature model version, which is added to the local vote result set and propagated to all other sites. Once cast, a vote is final and cannot be taken back, thus the vote results are a grow-only set that does not require any synchronization [48]. After a vote is processed at a site, a resolution criterion decides whether the voting phase is complete. For instance, such a criterion may involve plurality, majority or a consensus among all collaborators. When the voting phase is complete and there is a resolution, we compute the elected version from the vote result set and unfreeze the site, concluding the conflict resolution process. Otherwise, if voting is complete, but no resolution was achieved, the voting phase is restarted.

# 7 PRELIMINARY RESULTS

Although we have yet to evaluate our technique, we can already report preliminary results. Regarding the formal correctness of our technique, we can show that our technique complies with the CCI model, which is an established consistency model in literature [52, 53]. By reasoning about formal correctness, we are confident that our system allows highly-responsive, unconstrained collaboration, while still ensuring basic consistency properties.

Further, as a proof-of-concept, we have implemented our technique for collaborative, real-time feature modeling in the open-source prototype *variED*.[2] This web-based feature-modeling tool allows for real-time collaboration in a web browser and may serve as a basis for future user studies.

# 8 RELATED WORK

Closely related to our work is the *CoFM* environment that has been proposed by Yi et al. [57, 58]. With CoFM, stakeholders can construct a shared feature model and evaluate each other's work by selecting or denying model elements, resulting in a personal view for every collaborator. Our technique differs in that we only consider a single feature model, which is synchronized among all collaborators. Furthermore, we describe how to detect and resolve conflicting operations, which are not considered by CoFM. In addition, we employ optimistic replication to hide network latency, whereas CoFM uses a pessimistic approach.

Other works on feature-model editing have mostly focused on the single-user case [1, 8, 28, 35, 36]. To the best of our knowledge, none of these tools or techniques supports real-time collaboration. Rather, they allow asynchronous collaboration with version or variation control systems.

Linsbauer et al. [31] classify variation control systems, in which they notice a general lack of collaboration support compared to regular version control systems. In particular, Schwägerl and Westfechtel [45, 46] propose *SuperMod*, a variation control system for filtered editing in model-driven SPLs that supports asynchronous multi-user collaboration. However, SuperMod does not allow real-time editing and does not address conflicts that arise from the interaction of multiple collaborators.

Botterweck et al. [10] introduce *EvoFM*, a technique for modeling variability over time. Their catalog of evolution operators resembles the COs we presented in Section 3, but they do not explicitly address collaboration. Similarly, Nieke et al. [39, 40] encode the evolution of an SPL in a temporal feature model to guarantee valid configurations. With their technique, inconsistencies and evolution paradoxes can be detected. However, they do not address collaboration and provide no particular conflict resolution strategy. Change impact analyses on feature models have been proposed to identify and evaluate conflict potential of modeling decisions [17, 27, 32, 41]. These techniques do not explicitly address collaboration, but may guide collaborators in understanding and resolving conflicts.

# 9 CONCLUSION

In this paper, we presented a technique for collaborative, real-time feature modeling. Based on the general conditions of our considered use case scenarios, we defined requirements that such a technique should fulfill. Further, we described a technique for collaborative, real-time feature modeling that relies on operation-based editing and introduced primitive and compound operations. We extended the MVMD technique by introducing suitable conflict relations and a conflict resolution strategy that are suitable for feature modeling. In addition, we reported some preliminary experiences, showing the feasibility of our technique by implementing a prototype.

In future work, we want to conduct user studies to evaluate our technique. We also aim to address the question how to raise awareness of collaborators for potentially-conflicting editing operations in order to avoid conflicts in the first place.

# ACKNOWLEDGMENTS

The work of Elias Kuiter, Sebastian Krieter, and Jacob Krüger has been supported by the pure-systems Go SPLC 2019 Challenge.

---

[2] Sources, demonstration, and information: https://github.com/ekuiter/variED

## REFERENCES

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2013. FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming* 78, 6 (2013), 657–681.

[2] A. V. Aho, M. R. Garey, and J. D. Ullman. 1972. The Transitive Reduction of a Directed Graph. *SIAM J. Comput.* 1, 2 (1972), 131–137.

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

[4] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Grösslinger, and Dirk Beyer. 2013. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proceedings of the International Conference on Software Engineering.* IEEE, 482–491.

[5] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–636.

[6] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wasowski. 2014. Three Cases Of Feature-based Variability Modeling In Industry. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems.* Springer, 302–319.

[7] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems.* ACM, 7:1–7:8.

[8] Danilo Beuche. 2008. Modeling and Building Software Product Lines with Pure::Variants. In *Proceedings of the International Software Product Line Conference.* IEEE, 358–358.

[9] Sumeer Bhola, Guruduth Banavar, and Mustaque Ahamad. 1998. Responsiveness and Consistency Tradeoffs in Interactive Groupware. In *Proceedings of the Conference on Computer-Supported Cooperative Work.* ACM, 79–88.

[10] Goetz Botterweck, Andreas Pleuss, Deepak Dhungana, Andreas Polzer, and Stefan Kowalewski. 2010. EvoFM: Feature-Driven Planning of Product-Line Evolution. In *Proceedings of the International Workshop on Product Line Approaches in Software Engineering.* ACM, 24–31.

[11] T. F. Bowen, F. S. Dworack, C. H. Chow, N. Griffeth, G. E. Herman, and Y.-J. Lin. 1989. The Feature Interaction Problem in Telecommunications Systems. In *Proceedings of the International Conference on Software Engineering for Telecommunication Switching Systems.* IET, 59–62.

[12] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks* 41, 1 (2003), 115–141.

[13] Jeffrey Dennis Campbell. 2000. *Consistency Maintenance for Real-Time Collaborative Diagram Development.* Ph.D. Dissertation. University of Pittsburgh.

[14] David Chen. 2001. *Consistency Maintenance in Collaborative Graphics Editing Systems.* Ph.D. Dissertation. Griffith University.

[15] David Chen and Chengzheng Sun. 2001. Optional Instant Locking in Distributed Collaborative Graphics Editing Systems. In *Proceedings of the International Conference on Parallel and Distributed Systems.* IEEE, 109–116.

[16] Lianping Chen and Muhammad Ali Babar. 2011. A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. *Information and Software Technology* 53, 4 (2011), 344–362.

[17] Hyun Cho, Jeff Gray, Yuanfang Cai, Sonny Wong, and Tao Xie. 2011. Model-Driven Impact Analysis of Software Product Lines. In *Model-Driven Domain Analysis and Software Development: Architectures and Functions*, Janis Osis and Erika Asnina (Ed.). IGI Global, Chapter 13, 275–303.

[18] Krzysztof Czarnecki. 2013. Variability in Software: State of the Art and Future Directions. In *Fundamental Approaches to Software Engineering (FASE).* Springer, 1–5.

[19] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems.* ACM, 173–182.

[20] Gabriele D'Angelo, Angelo Di Iorio, and Stefano Zacchiroli. 2018. Spacetime Characterization of Real-Time Collaborative Editing. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 41:1–41:19.

[21] Alan R. Dennis, Sridar K. Pootheri, and Vijaya L. Natarajan. 1998. Lessons from the Early Adopters of Web Groupware. *Journal of Management Information Systems* 14, 4 (1998), 65–86.

[22] Prasun Dewan, Rajiv Choudhary, and Honghai Shen. 1994. An Editing-Based Characterization of the Design Space of Collaborative Applications. *Journal of Organizational Computing* 4, 3 (1994), 219–239.

[23] Clarence Ellis, Simon Gibbs, and Gail Rein. 1991. Groupware: Some Issues and Experiences. *Commun. ACM* 34, 1 (1991), 39–58.

[24] Thomas Fogdal, Helene Scherrebeck, Juha Kuusela, Martin Becker, and Bo Zhang. 2016. Ten Years of Product Line Engineering at Danfoss: Lessons Learned and Way Ahead. In *Proceedings of the International Software Product Line Conference (SPLC).* ACM, 252–261.

[25] Simon Gibbs. 1989. LIZA: An Extensible Groupware Toolkit. In *Proceedings of the Conference on Human Factors in Computing Systems.* ACM, 29–35.

[26] Saul Greenberg and David Marwood. 1994. Real Time Groupware as a Distributed System: Concurrency Control and itss Effect on the Interface. In *Proceedings of the Conference on Computer-Supported Cooperative Work.* ACM, 207–217.

[27] Ines Hajri, Arda Goknil, Lionel C. Briand, and Thierry Stephany. 2018. Change Impact Analysis for Evolving Configuration Decisions in Product Line Use Case Models. *Journal of Systems and Software* 139 (2018), 211–237.

[28] Charles W. Krueger. 2007. BigLever Software Gears and the 3-Tiered SPL Methodology. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications.* ACM, 844–845.

[29] Elias Kuiter. 2019. *Consistency Maintenance for Collaborative Real-Time Feature Modeling.* Bachelor Thesis. University of Magdeburg.

[30] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[31] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *Proceedings of the International Conference on Generative Programming and Component Engineering.* ACM, 49–62.

[32] Jihen Maâzoun, Nadia Bouassida, and Hanêne Ben-Abdallah. 2016. Change Impact Analysis for Software Product Lines. *Journal of King Saud University — Computer and Information Sciences* 28, 4 (2016), 364–380.

[33] Christian Manz, Michael Stupperich, and Manfred Reichert. 2013. Towards Integrated Variant Management In Global Software Engineering: An Experience Report. In *International Conference on Global Software Engineering.* IEEE, 168–172.

[34] Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms.* North Holland, 215–226.

[35] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE.* Springer.

[36] Marcilio Mendonça, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: Software Product Lines Online Tools. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications.* ACM, 761–762.

[37] Meredith Ringel Morris, Kathy Ryall, Chia Shen, Clifton Forlines, and Frederic Vernier. 2004. Beyond "Social Protocols": Multi-user Coordination Policies for Co-Located Groupware. In *Proceedings of the Conference on Computer-Supported Cooperative Work.* ACM, 262–265.

[38] Damir Nešić, Jacob Krüger, Stefan Stănciulescu, and Thorsten Berger. 2019. Principles of Feature Modeling. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* ACM.

[39] Michael Nieke, Christoph Seidl, and Sven Schuster. 2016. Guaranteeing Configuration Validity in Evolving Software Product Lines. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems.* ACM, 73–80.

[40] Michael Nieke, Christoph Seidl, and Thomas Thüm. 2018. Back to the Future: Avoiding Paradoxes in Feature-Model Evolution. In *Proceedings of the International Software Product Line Conference.* ACM, 48–51.

[41] Paulius Paskevicius, Robertas Damasevicius, and Vytautas Štuikys. 2012. Change Impact Analysis of Feature Models. In *Proceedings of the International Conference on Information and Software Technologies.* Springer, 108–122.

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

[43] Atul Prakash. 1999. Group Editors. In *Computer Supported Co-Operative Work*, Michel Beaudouin-Lafon (Ed.). Wiley, Chapter 5, 103–134.

[44] Rüdiger Schollmeier. 2001. A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. In *Proceedings of the International Conference on Peer-to-Peer Computing.* IEEE, 101–102.

[45] Felix Schwägerl and Bernhard Westfechtel. 2016. Collaborative and Distributed Management of Versioned Model-Driven Software Product Lines. In *International Joint Conference on Software Technologies.* SciTePress, 83–94.

[46] Felix Schwägerl and Bernhard Westfechtel. 2017. Maintaining Workspace Consistency in Filtered Editing of Dynamically Evolving Model-Driven Software Product Lines. In *Proceedings of the International Conference on Model-Driven Engineering and Software Development.* SciTePress, 15–28.

[47] Reinhard Schwarz and Friedemann Mattern. 1994. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing* 7, 3 (1994), 149–174.

[48] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A Comprehensive Study of Convergent and Commutative Replicated Data Types.* Research Report RR-7506. Inria – Centre Paris-Rocquencourt.

[49] Mark Stefik, Gregg Foster, Daniel G. Bobrow, Kenneth Kahn, Stan Lanning, and Lucy Suchman. 1987. Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings. *Commun. ACM* 30, 1 (1987), 32–47.

[50] Chengzheng Sun and David Chen. 2000. A Multi-Version Approach to Conflict Resolution in Distributed Groupware Systems. In *Proceedings of the International Conference on Distributed Computing Systems.* IEEE, 316–325.

[51] Chengzheng Sun and David Chen. 2002. Consistency Maintenance in Real-Time Collaborative Graphics Editing Systems. *ACM Transactions on Computer-Human Interaction* 9, 1 (2002), 1–41.

[52] Chengzheng Sun and Clarence Ellis. 1998. Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. In *Proceedings of the*

*Conference on Computer-Supported Cooperative Work*. ACM, 59–68.

[53] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. 1998. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction* 5, 1 (1998), 63–108.

[54] Chengzheng Sun, Steven Xia, David Sun, David Chen, Haifeng Shen, and Wentong Cai. 2006. Transparent Adaptation of Single-User Applications for Multi-User Real-Time Collaboration. *ACM Transactions on Computer-Human Interaction* 13, 4 (2006), 531–582.

[55] Volker Wulf. 1995. Negotiability: A Metafunction to Tailor Access to Data in Groupware. *Behaviour & Information Technology* 14, 3 (1995), 143–151.

[56] Liyin Xue, Mehmet Orgun, and Kang Zhang. 2003. A Multi-Versioning Algorithm for Intention Preservation in Distributed Real-Time Group Editors. In *Proceedings of the Australasian Computer Science Conference*. ACS, 19–28.

[57] Li Yi, Wei Zhang, Haiyan Zhao, Zhi Jin, and Hong Mei. 2010. CoFM: A Web-Based Collaborative Feature Modeling System for Internetware Requirements' Gathering and Continual Evolution. In *Proceedings of the Asia-Pacific Symposium on Internetware*. ACM, 23:1–23:4.

[58] Li Yi, Haiyan Zhao, Wei Zhang, and Zhi Jin. 2012. CoFM: An Environment for Collaborative Feature Modeling. In *Proceedings of the International Requirements Engineering Conference*. IEEE, 317–318.