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:

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.
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 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 \( (F, C) \) where \( F \) is a finite set of features and \( C \) is a finite set of cross-tree constraints.\(^1\)

A feature \( F \in FM \) is a tuple \( (ID, \text{parentID}, \text{optional}, \text{groupType}, \text{abstract}, \text{name}) \) where \( ID \in ID \), parentID \( \in ID \cup \{\perp, \top\} \), optional \( \in \{\text{true, false}\} \), groupType \( \in \{\text{and, or, alternative}\} \), abstract \( \in \{\text{true, false}\} \), and name is a string.

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

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

The set \( ID \) contains all Universally Unique Identifiers (UUIDs) that can be used within a feature model. \( \perp \) denotes the parentID of the root feature, while \( \top \) denotes the parentID of an uninitialized feature.

To simplify our definitions, we declare the two sets \( FM.FID \) and \( FM.CID \) 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

- \( FM.FID = \{F.ID | F \in FM.F\} \),
- \( FM.CID = \{C.ID | C \in FM.C\} \), and
- \( \leq_{FM} \) as the reflexive transitive closure of \( \{(A.ID, B.ID) | A \in FM.F, B.ID \in FM.FID \cup \{\perp, \top\} \land A.parentID = B.ID\} \).

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: \( |FM.F| = |FM.FID| \land |FM.C| = |FM.CID| \)
- Valid parents: \( \forall F \in FM.F : F.parentID \in FM.FID \cup \{\perp, \top\} \)
- Valid constraints: \( \forall C \in FM.C : \text{Var}(C, \phi) \subseteq FM.FID \)
- Single root: \( \exists F \in FM.F : F.parentID = \perp \)
- Acyclic: \( \forall F \in FM.F : F.ID \leq_{FM} \perp \lor F.ID \leq_{FM} \top \)

We denote the set of all legal feature models as \( FM \).

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 adopt 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 \( O_b \)) if and only one of the following is true:

- \( i = j \) and \( O_a \) is generated before \( O_b \)
A collaborative feature model editor must support a variety of operations, including creating, removing, and modifying features and cross-tree constraints. For each operation, we provide formal requirements that enable our technique and its integration. 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:

**Definition 5.** Let FM ∈ FFM. Further, let PO and CO be a primitive and a compound operation whose preconditions are satisfied with regard to FM. Then, FM’ = applyPO(FM, PO) denotes the feature model FM’ that results from applying PO to FM. Further, we define 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 applyPO 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 always preserve the legality of feature models (cf. Definition 3) in a single-user scenario [29].
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.

Req2: 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.

Req3: 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, multiversioning 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 ⊗ 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 \( O_b \), \( O_a \not\otimes O_b \Rightarrow O_a \otimes O_b \).

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 \( E = \{(O_a, O_b) \mid O_a, O_b \in GO \land O_a \rightarrow O_b \} \) is the set of edges. Then, the set of causally preceding operations for an \( O \in GO \) is defined as \( CPG(O) := \{O_a \mid (O_a, O) \in E\} \).

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 \( CIPG(O) := \{O_a \mid (O_a, O) \in E'\} \).

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 \( O_a \rightarrow O_x \rightarrow O_b \). 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 \( \Box \)) serves to guarantee this property. It may be computed with \texttt{outerConflicting}, a recursive algorithm that uses the CDAG to propagate detected conflicts to all causally succeeding operations:

\begin{verbatim}
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 \parallel CO_b \lor CO_a = CO_b then return false
if \exists CIPO_a \in CIPO(G, CO_a, CIPO_b) \lor \exists CIPO_b \in CIPO(G, CO_b, CIPO_a)
then return true
return CO_a \Box CO_b
end function
\end{verbatim}

In the basic case, \texttt{outerConflicting} defers the conflict detection to the inner conflict relation (\( \bigotimes \)). The outer 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 \texttt{outerConflicting}, we can compute \( \Box \) as follows:

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

5.3 Topological Sorting Strategy

The outer conflict relation \( \Box \) ensures the existence of a suitable feature model. To actually produce such a model, we use \texttt{applyCOs}(G, FM, COs) := reduce(applyCO, FM, topsort(COs, G)) where \texttt{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 \texttt{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 \( \bigotimes \) detects conflicts that are specific to feature modeling. We introduce \texttt{innerConflicting} to determine \( \bigotimes \) for two given compound operations:

\begin{verbatim}
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 \parallel CO_b \lor CO_a = CO_b then return false
if syntacticallyConflicting(G, FM, CO_a, CO_b)
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 SP; SP(FM) = true
end function
\end{verbatim}

This algorithm makes use of \texttt{syntacticallyConflicting}, which determines whether two COs have a **syntactic conflict** that concerns basic syntactic properties of feature models. \texttt{syntacticallyConflicting} does so by applying both operations to a suitable feature model derived with \texttt{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 \( \bigotimes \), as required by Definition 6, \texttt{innerConflicting} uses \texttt{syntacticallyConflicting} to check for syntactic conflicts in both directions. Finally, \texttt{innerConflicting} may check additional arbitrary semantic properties on a feature model that includes the effects of \( CO_a \) and \( CO_b \). A semantic property \( SP \in SP \) is a deterministic function \( SP: FM \rightarrow \{true, false\} \) 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 \texttt{innerConflicting}, we can compute \( \bigotimes \) as follows:

**Definition 9.** Two compound operations \( CO_a \) and \( CO_b \) are in inner conflict, i.e., \( CO_a \bigotimes CO_b \), iff \texttt{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 \texttt{innerConflicting}.

Our conflict detection technique can now be implemented by using \( \bigotimes \) 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].
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 opensource prototype variED. 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.


