Empirical Software Engineering (2021) 26:24
https://doi.org/10.1007/s10664-020-09892-x

variED: an editor for collaborative, real-time feature modeling

Elias Kuiter[1] · Sebastian Krieter[1,2] · Jacob Krüger[1,3] · Gunter Saake[1] · Thomas Leich[2]

Accepted: 27 November 2020Published online: 02 March 2021

Abstract
Feature models are a helpful means to document, manage, maintain, and configure the variability of a software system, and thus are a core artifact in software product-line engineering. Due to the various purposes of feature models, they can be a cross-cutting concern in an organization, integrating technical and business aspects. For this reason, various stakeholders (e.g., developers and consultants) may get involved into modeling the features of a software product line. Currently, collaboration in such a scenario can only be done with face-to-face meetings or by combining single-user feature-model editors with additional communication and version-control systems. While face-to-face meetings are often costly and impractical, using version-control systems can cause merge conflicts and inconsistency within a model, due to the different intentions of the involved stakeholders. Advanced tools that solve these problems by enabling collaborative, real-time feature modeling, analogous to Google Docs or Overleaf for text editing, are missing. In this article, we build on a previous paper and describe (1) the extended formal foundations of collaborative, real-time feature modeling, (2) our conflict resolution algorithm in more detail, (3) proofs that our formalization converges and preserves causality as well as user intentions, (4) the implementation of our prototype, and (5) the results of an empirical evaluation to assess the prototype's usability. Our contributions provide the basis for advancing existing feature-modeling tools and practices to support collaborative feature modeling. The results of our evaluation show that our prototype is considered helpful and valuable by 17 users, also indicating potential for extending our tool and opportunities for new research directions.

Keywords Software product lines · Groupware · Feature modeling · Variability · Consistency maintenance · Collaboration

Communicated by: Laurence Duchien, Thomas Thüm and Paul Grünbacher

This paper has been awarded the Empirical Software Engineering (EMSE) open science badge.

This article belongs to the Topical Collection: *Configurable Systems*

This article belongs to the Topical Collection: *Open Science*

✉ Sebastian Krieter
skrieter@hs-harz.de

Extended author information available on the last page of the article.

1 Introduction

Modeling the variability of a configurable platform (i.e., a Software Product Line (SPL)) is essential for an organization to document and manage all implemented *software features* and to derive valid product configurations that are tailored to different customer requirements (Apel et al. 2013a; Pohl et al. 2005; Czarnecki et al. 2012; Berger et al. 2015). The variability model of an SPL is not limited to representing implementation artifacts, but can incorporate additional artifacts, such as requirements, documentation, and tests (Berger et al. 2013; Czarnecki 2013). Moreover, while variability models are usually structured according to design decisions specific to the SPL's domain, they can also provide a more abstract representation for other stakeholders (Nešić et al. 2019). For instance, a variability model may be used to allow end users to configure their own products. To derive a meaningful variability model that satisfies the defined needs (e.g., representing solution or problem space), all relevant stakeholders must work collaboratively (Berger et al. 2014; Nešić et al. 2019).

Various tools have been proposed to facilitate variability modeling with tailored user interfaces, automated analyses, and other editing support. Established examples in the commercial domain are *pure::variants* (Beuche 2008) and *Gears* (Krueger 2007); as well as several research prototypes (Alves Pereira et al. 2014; Horcas et al. 2019). Still, to the best of our knowledge, there is neither a tool nor a technique that supports *collaborative, real-time editing* of the same variability model, similar to the text editors *Google Docs* and *Overleaf*. As far as we know, existing tools allow only a single user to edit the model, and multiple users have to rely on direct communication or version-control systems to collaborate. Both of these strategies can have major drawbacks that hamper efficient collaboration during the modeling process. Direct communication requires either locality of all stakeholders or screen-sharing, and does not allow all stakeholders to edit the model (e.g., to sketch a solution or to work on different parts of the model). Version-control systems, such as Git, allow the stakeholders to share and edit a variability model at different places and at the same time. However, version-control systems are not designed for real-time editing, which may easily cause merge conflicts, not only syntactically, but also semantically (e.g., merged changes may result in an invalid feature model, dead features, or broken dependencies). Collaborative, real-time variability modeling promises advantages in several use cases, such as:

- 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).
- Domain engineers can share and discuss the variability model with other domain experts, allowing them to evolve the model based on 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 notation for variability models are *feature models* and their visual representations, *feature diagrams* (Czarnecki et al. 2012; Berger et al. 2013; Chen and Babar 2011; Nešić et al. 2019). For this reason, we focus on the collaborative editing of feature diagrams in this article. However, our general technique can be applied to other representations of variability models as well.

In this article, we extend a previous paper (Kuiter et al. 2019b), in which we describe the conceptual foundations of collaborative, real-time feature modeling. Within that paper, we define requirements that our technique aims to fulfill, derive formal specifications for

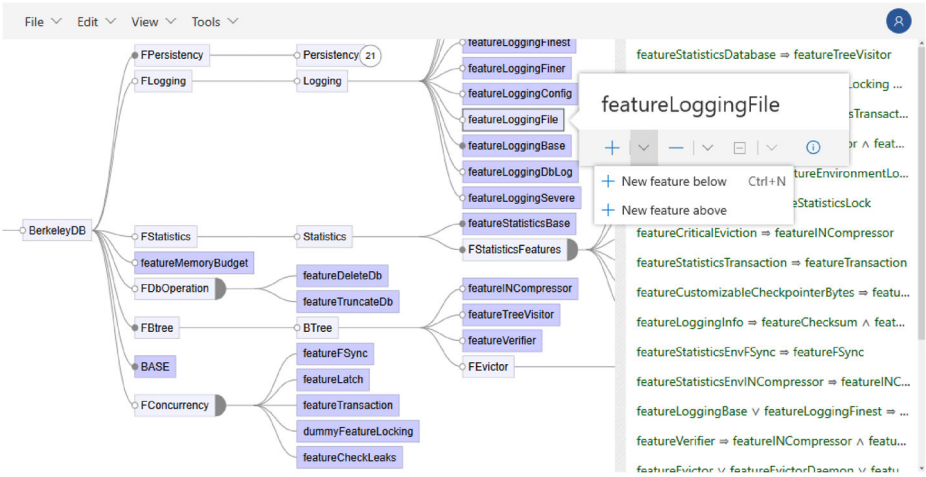


Fig. 1 Editing a feature model inside our web-based modeling tool variED

the operations that we need to develop, sketch how to resolve conflicts, and implement and verify a corresponding tool for collaborative feature modeling (cf. Figure 1). To extend that paper, we include more details and new information on our specifications, algorithms, and their implementation, resulting in the first complete description of our technique, including all operations, conflict detection, and conflict resolution. We further present new contributions by showing the correctness of our technique, adding a description of our tool, and providing an empirical evaluation in the form of a user study with 17 participants. Consequently, the overall focus in this article is not on theoretical foundations, as in our previous paper. Instead, we focus on the concrete implementation of our technique and its evaluation. Overall, we make the following contributions:

1. We extend the formal foundations of collaborative, real-time feature modeling.
2. We report the algorithms, requirements, and techniques that are part of our own technique and describe our corresponding tool implementation.
3. We show that our technique works correctly in terms of concurrency control.
4. We report upon a user study to evaluate the usability of our tool.
5. We provide public GitHub repositories that include the open-source implementation of our tool[1] as well as our questionnaire and anonymized responses.[2]

The goal of our technique is to support use cases that fulfill three general conditions. First, we assume that different stakeholders want to work on the same feature model at the same time, for example, to coordinate independent or interacting tasks (Manz et al. 2013), which requires techniques for concurrency control. Second, based on insights from real-world case studies and established practices (Fogdal et al. 2016; Berger et al. 2014; Nešić et al. 2019), we assume that a small team (i.e., ten or fewer stakeholders) edits and maintains a feature model. Considering larger teams, collaboration and automatic conflict resolution become much more challenging. Finally, we assume that not all stakeholders are co-located, but

[1] https://doi.org/10.5281/zenodo.4259912
[2] https://doi.org/10.5281/zenodo.4259914

remotely connected (Manz et al. 2013). For this reason, we aim to support peer-to-peer as well as client-server architectures (Schollmeier 2001).

The remainder of this article is structured as follows. In Section 2, we introduce the formal foundations of feature models and their properties, and describe the general concept of collaborative, real-time editing. We present the details of our technique, including the operation model, conflict detection, and conflict resolution in Section 3. Then, in Section 4, we show the correctness of our technique using formal proofs before explaining the details of our implementation in Section 5. In Section 6, we present the design and results of our user study. Finally, we present related work in Section 7 and conclude our findings in Section 8.

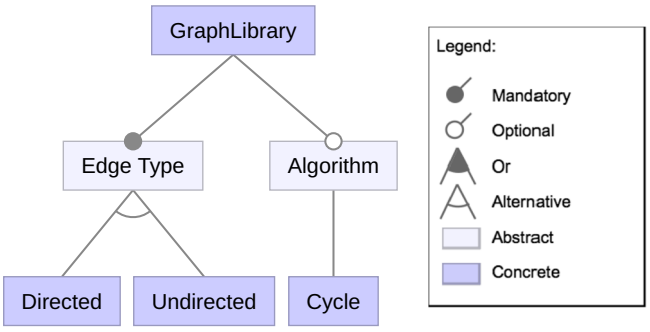
2 Foundations

In this section, we present the background and formal concepts of feature modeling and collaborative, real-time editing.

2.1 Feature Modeling

Feature modeling is a domain-engineering activity, during which an organization defines and documents the desired variability of its SPL (Apel et al. 2013a; Pohl et al. 2005; Krüger et al. 2020). This activity results in a feature model, which expresses variability in the notion of features and dependencies between these features (e.g., one feature may require or exclude another one). Features typically represent a user-visible functionality that can be either present or absent in a concrete product of the SPL. Several notations for feature models have been proposed in the literature, including feature diagrams, grammars, and propositional formulas (Batory 2005; Berger et al. 2013; Chen and Babar 2011; Czarnecki et al. 2012; Schobbens et al. 2006).

In this article, we focus on feature diagrams (cf. Figure 1 for a larger example in variED), which are visual representations of feature models as tree structures. This notation is human-readable, established in practice (Berger et al. 2013; Nešić et al. 2019), and well-supported by feature-modeling tools (Meinicke et al. 2017; Beuche 2008; Krueger 2007). We consider *basic* feature diagrams, which use binary features and constraints that can be



$$Cycle \Rightarrow Directed$$

Fig. 2 A simplified feature diagram of an SPL of graphs and their algorithms

expressed as propositional formulas. In contrast, *extended* feature diagrams allow organizations to add additional attributes to features and to use constraints of higher-order logic. As an example, we display an excerpt of a feature diagram in Fig. 2, which represents an SPL of graph algorithms. The feature model specifies that every product must contain the mandatory feature `Edge Type` and that each product's `Edge Type` must either be `Directed` or `Undirected`. Further, a product may include the optional feature `Algorithm`, which requires the feature `Cycle`. The cross-tree constraint below the tree defines that any product comprising the feature `Cycle` must also include `Directed`. In contrast to *concrete* features, the *abstract* features `Edge Type` and `Algorithm` do not map to implementation artifacts of the underlying SPL, but are used for structuring the feature model.

To allow us to precisely define our technique for collaborative feature modeling, we formalize feature models as follows:

Definition 1 (Feature Model)

- The *identifier pool* $\boldsymbol{ID}$ is an infinite set of strings that may be used within a feature model to identify features or cross-tree constraints.
- A *feature* is a tuple (ID, $parentID$, $optional$, $groupType$, $abstract$, $name$) where
  - $ID \in \boldsymbol{ID}$,
  - $parentID \in \boldsymbol{ID} \cup \{\bot, \dagger\}$,
  - $optional \in \{true, false\}$,
  - $groupType \in \{and, or, alternative\}$,
  - $abstract \in \{true, false\}$, and
  - $name$ is any string.
- A *cross-tree constraint* is a tuple (ID, $parentID$, ϕ) where $ID \in \boldsymbol{ID}$, $parentID \in \{\bot, \dagger\}$, and ϕ is a propositional formula with variables ranging over $\boldsymbol{ID}$, that is, $Var(\phi) \subseteq \boldsymbol{ID}$.
- A *feature model* FM then is a tuple ($\boldsymbol{F}$, $\boldsymbol{C}$) where $\boldsymbol{F}$ is a finite set of features and $\boldsymbol{C}$ is a finite set of cross-tree constraints.

In this definition, $\bot$ denotes the $parentID$ of the root feature and of all visible constraints, while $\dagger$ denotes the $parentID$ of removed features and constraints (cf. Definition 4). Our definition captures the properties of basic feature diagrams and integrates well with our technique for collaborative feature modeling. In particular, we introduce the *identifier pool* $\boldsymbol{ID}$, which is critical for describing the feature-modeling operations needed. This identifier pool and our formalization of the tree structure sets our definition apart from previous formalizations by Schobbens et al. (2007) and Durán et al. (2017), and makes it suitable for our concurrency-control technique.

Example 1 (Feature Model) We use Definition 1 to describe the feature diagram in Fig. 2. Let $FM = (\boldsymbol{F}, \boldsymbol{C})$ where

$$
\begin{aligned}
\boldsymbol{F} = \{&(GPL, \bot, \underline{false}, and, false, \texttt{GraphLibrary}),\\
&(ET, GPL, false, alternative, true, \texttt{Edge Type}),\\
&(Dir, ET, \underline{true}, \underline{and}, false, \texttt{Directed}),\\
&(Undir, ET, \underline{true}, \underline{and}, false, \texttt{Undirected}),\\
&(Alg, GPL, true, or, true, \texttt{Algorithm}),\\
&(Cyc, Alg, \underline{true}, \underline{and}, false, \texttt{Cycle})\}\\
\text{and } \boldsymbol{C} = \{&(constraint1, \bot, Cyc \Rightarrow Dir)\}.
\end{aligned}
$$

We use arbitrary identifiers in this example; in practice they will be generated automatically to ensure that each identifier is unique. The `GPL` feature is the root feature, as its *parentID* is $\perp$. Every other feature has a *parentID* that identifies its parent in the feature tree. We underline values that are effectively ignored by the user interface. For example, the `Directed` feature is part of an *alternative* group (defined in its parent `Edge Type`), which overrides its optional flag. However, it is advantageous to include this information in the formalization to facilitate conflict detection.

We remark that we introduce a dot notation to access a tuple's elements (which we also refer to as *attributes*) in order to improve the readability of our formalizations. For example, $FM.\boldsymbol{F}$ refers to the first element of $FM = (\boldsymbol{F}, \boldsymbol{C})$: the set of features $\boldsymbol{F}$ contained in FM. Further, in several instances, we interpret the sets $\boldsymbol{F}$ and $\boldsymbol{C}$ as functions $\boldsymbol{ID} \mapsto \boldsymbol{F}$ and $\boldsymbol{ID} \mapsto \boldsymbol{C}$ to look up features and cross-tree constraints by their *ID* in the feature model (if defined uniquely). So, in Example 1, $FM.\boldsymbol{F}(Dir).parentID$ refers to the value *ET* defined in the *parentID* field for the feature identified by *Dir* in *FM*.

To simplify our algorithms and proofs, we define two sets $\boldsymbol{F}_{FM}^{ID}$ and $\boldsymbol{C}_{FM}^{ID}$ for feature and constraint identifiers, as well as the parent-child relation *descends from* ($\preceq_{FM}$):

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

- $\boldsymbol{F}_{FM}^{ID} \coloneqq \{F.ID \mid F \in FM.\boldsymbol{F}\}$, that is, the feature IDs defined in FM,
- $\boldsymbol{C}_{FM}^{ID} \coloneqq \{C.ID \mid C \in FM.\boldsymbol{C}\}$, that is, the cross-tree constraint IDs defined in FM, and
- the relation $\preceq_{FM}$ as the reflexive transitive closure of $\{(A.ID, B^{ID}) \mid A \in FM.\boldsymbol{F}, B^{ID} \in \boldsymbol{F}_{FM}^{ID} \cup \{\perp, \dagger\} \wedge A.parentID = B^{ID}\}$, that is, a feature *descends from* another when it is an immediate or indirect child.

Using our notation from Definition 1, we can formally describe any feature model based on its feature-diagram representation. However, this definition allows that integral properties of feature models may be violated (e.g., a feature model must not contain cycles). These properties are important, as we intend to manipulate feature models by means of operations. Consequently, we also define the following conditions to describe *legal* feature models (such as Example 1):

Definition 3 A feature model FM is considered *legal* if and only if all of the following conditions are true:

- All identifiers are unique: $\left|\boldsymbol{F}_{FM}^{ID}\right| \cap \left|\boldsymbol{C}_{FM}^{ID}\right| = \varnothing \wedge |FM.\boldsymbol{F}| = \left|\boldsymbol{F}_{FM}^{ID}\right| \wedge |FM.\boldsymbol{C}| = \left|\boldsymbol{C}_{FM}^{ID}\right|$
- All parent identifiers refer to valid features: $\forall F \in FM.\boldsymbol{F}\colon F.parentID \in \boldsymbol{F}_{FM}^{ID} \cup \{\perp, \dagger\}$
- All variables in constraints refer to valid features: $\forall C \in FM.\boldsymbol{C}\colon Var(C.\phi) \subseteq \boldsymbol{F}_{FM}^{ID}$
- There exists exactly one root feature: $\exists! F \in FM.\boldsymbol{F}\colon F.parentID = \perp$
- The feature graph is acyclic and, thus, a tree: $\forall F^{ID} \in \boldsymbol{F}_{FM}^{ID}\colon F^{ID} \preceq_{FM} \perp \vee F^{ID} \preceq_{FM} \dagger$

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

Naturally, our technique has to allow operations for removing features and constraints from a feature model. However, since our technique should also be extensible for *undo* and *redo* operations, we cannot simply remove an element from $\boldsymbol{F}$ or $\boldsymbol{C}$, but need to keep track of its old state and position. To achieve this, instead of removing an element entirely, we move it to a separate, non-visible area that we call *graveyard*. So, we adapt the *tombstone* concept used in distributed systems, where removed entities are still kept in memory to

allow undoing and rejecting changes (Chen and Sun 2001c; Oster et al. 2006; Shapiro et al. 2011).

Definition 4 Let $FM \in \boldsymbol{FM}$.

- A feature $F \in FM.\boldsymbol{F}$ is *graveyarded*, denoted as $F.ID \in \boldsymbol{F}^{\dagger}_{FM}$, if and only if $F.ID \preceq_{FM} \dagger$ (where $\dagger$ can be thought of as spanning a second, invisible feature tree).
- A cross-tree constraint $C \in FM.\boldsymbol{C}$ is *graveyarded*, denoted as $C.ID \in \boldsymbol{C}^{\dagger}_{FM}$, if and only if $C.parentID = \dagger \vee \exists F^{ID} \in Var(C.\phi) \colon F^{ID} \in \boldsymbol{F}^{\dagger}_{FM}$ (i.e., it has been ex- or implicitly removed).

We utilize this formalization of feature models to design our collaborative operation model (cf. Section 3.1) and conflict detection algorithms (cf. Section 3.2).

2.2 Collaborative, Real-Time Editing

The research domain of Computer-Supported Cooperative Work (CSCW) is concerned with collaborative activities and their computer-assisted coordination (Carstensen and Schmidt 1999; Grudin 1994). CSCW systems (also called groupware Ellis et al. 1991) are typically classified according to the time and location dimensions (Baecker et al. 1995; Johansen 1991). Collaboration may happen *synchronously* (i.e., at the same time) or *asynchronously* (i.e., at different times). Analogously, collaborators may work *co-located* (i.e., at the same location) or *remotely* (i.e., at different locations). We are interested in synchronous, remote collaboration, as existing feature-modeling tools do not support this combination, yet. For this reason, we design a collaborative, real-time editor that enables shared editing by remotely connected users (Prakash 1999). We use an *operation-based editing* model to achieve change propagation and use the definitions of Sun et al. (1998) to formally describe *concurrency*, *conflicts*, and *consistency*.

2.2.1 Operation-Based Editing

A simple, but effective, technique for propagating changes through a network is to send the entire document of one user to all other users. However, this technique has drawbacks, as it sends lots of redundant data and makes consistency maintenance more complicated. Another technique of distributing changes is operation-based editing (Dewan 1999; Shapiro et al. 2011). Instead of sending complete copies of a modified document to other users, only an abstract operation is sent, allowing to transform the previous state of a document into the new state. So, the complete document must be sent only once: when initializing the system. We define an operation as follows:

Definition 5 (Operation) An *operation* is the description of an atomic manipulation of a document with a distinct user intention. It is applied to a document to transform it from an old to a new (modified) state.

To identify relevant operations for feature modeling, we reviewed two established single-user feature-modeling tools that we could investigate without charges, the academic *FeatureIDE* (Meinicke et al. 2017) and the industrial *pure::variants* (Beuche 2008). We can consider feature-model editors as object-based graphics editors, in which features (or possibly entire subtrees of features) and constraints are represented as objects. Feature-modeling

operations can target one or multiple objects based on their unique identifiers. To issue an operation, a user first creates a selection that includes all objects that operation should target, then proceeds with the actual operation. In Section 2.1, we defined feature models according to this operation model, as each feature and cross-tree constraint is assigned a unique identifier that can serve as an operation target. Thus, rather than identifying features via their name, we use the unique identifiers in the ***ID*** set and represent a feature's name as an additional attribute.

We classify operations on the feature model as view-, feature-, or constraint-related. First, *view-related operations* customize the visual representation of a feature model to a user's demands, for example, changing the diagram layout (e.g., tree, table, or graph visualization), visually pruning the feature tree, and hiding cross-tree constraints. As view-related operations do not change the semantics of the feature model (i.e., the modeled set of product variants), we do not consider them for collaborative feature modeling. Instead, we can apply view-related operations locally and provide each collaborator with their own, customized modeling view according to their preferences. Second, *feature-related operations* encompass creating, removing, and updating features in the feature tree. Users can create features at different positions within the feature tree, for instance, below or above another feature. They can also move features within the tree by moving a sub-tree below another feature. In addition, most attributes of a feature can be updated, including the *optional*, *groupType*, or any other arbitrary attribute, such as the name or description. Third, *constraint-related operations* allow users to create, edit, or remove arbitrary cross-tree constraints. These operations are similar to the feature-related ones, but they are simpler as they do not span a tree. As aforementioned, we focus on the last two classes of operations, since they introduce semantic changes that we need to propagate to users. In Section 3.1, we describe our operation model for collaborative feature modeling.

2.2.2 Concurrency and Conflicts

In collaborative editing systems, multiple users can create operations at different sites (i.e., workstations) at different times. To ensure that a document has the same state for all users, their sites must be synchronized eventually. However, as synchronization between sites is affected by network latency, and thus not instant, we cannot simply track the order of submitted operations based on physical time. Instead, we adapt a well-known strict partial order (Lamport 1978; Mattern 1988; Sun et al. 1998) to determine the temporal (and thus causal) relationships of operations, and define the notion of concurrency:

Definition 6 (Causal Ordering) 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 if at least one of the following is true:

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

where *before* refers to a site's local physical time. O_b is then said to *depend on* O_a. Further, O_a and O_b are *causally related* if and only if $O_a \rightarrow O_b$ or $O_b \rightarrow O_a$. Otherwise they are *concurrent*, denoted as $O_a \parallel O_b$.

This causal ordering replaces physical timestamps in the communication between different sites. Note that we make no distinction between time and causality: If $O_a \rightarrow O_b$, we say

that O_a causes O_b, even if the intention of O_a is not related to O_b at all. This is a convention in distributed systems to capture all events that *may have* caused an event, which facilitates concurrency control (Mattern 1988).

Two concurrent operations may cause a *conflict*, if applying both on a document would lead to an inconsistent state. Consequently, whether two operation are in conflict, depends on the kind of document. In Section 3.2, we define conflicts of feature-modeling operations.

2.2.3 Consistency

Several challenges hamper the maintenance of a consistent document state in collaborative, real-time editors (Sun and Chen 2002; Sun et al. 1998). First, operations in such editors do not generally commute, meaning that they do not yield the same result for all execution orders. So, the document state may *diverge* when concurrent operations are applied in varying orders at different sites. Second, when an operation arrives before an operation it depends on, the user interface may behave surprisingly and the document may become invalid. This challenge arises from network latency, which may lead to *causality violation* in peer-to-peer architectures. Third, concurrent operations may violate each other's *intention* (i.e., an operation's execution effect when applied on the document state from which it was generated). For example, two operations that set the same feature's name to different values are intention-violating, as both override the other operation's intention. Apart from these syntactic challenges, *semantic* consistency properties should also be considered (Sun et al. 1998). In Section 3.2, we identify semantic properties that we aim to ensure when editing feature models. Based on these challenges, Sun et al. (1996, 1998) have proposed a consistency model for collaborative, real-time editors as follows:

Definition 7 (CCI Model) A collaborative, real-time editing system is *CCI-consistent* if it always maintains all of the following properties:

- *Convergence:* When the same set of operations have been executed at all sites, all copies of the shared document are identical.
- *Causality Preservation:* For any pair of operations O_a and O_b, if $O_a \rightarrow O_b$, then O_a is executed before O_b at all sites.
- *Intention Preservation:* For any operation O, the effects of executing O at all sites are the same as the intention of O, and the effect of executing O does not change the effects of concurrent operations.

These properties define what behavior collaborators can expect from the system. In Section 4, we point out why our concept conforms to the CCI model. So, we ensure the syntactic and semantic consistency of edited feature models at all times. We proceed to describe the technique we adapt to build a CCI-consistent collaborative feature model editor.

2.2.4 Multi-Version Multi-Display Technique

To achieve CCI-consistency, several concurrency-control techniques have been proposed. Roughly, we can distinguish between techniques that work either on a single document or on multiple versions of a document to resolve conflicts. Single-document techniques are, for example, turn taking (Ellis et al. 1991; Greenberg 1991), locking (Elmasri and Navathe 2010; Greenberg and Marwood 1994), serialization (Berlage and Genau 1993; Greenberg and Marwood 1994), and operational transformation (Ellis and Gibbs 1989; Sun et al.

1998, 2004). However, these techniques have severe disadvantages for our considered use cases (Kuiter 2019a): None of these techniques can properly handle conflicting operations, as they usually block the document or its parts completely to avoid conflicts (they are *pessimistic*), or can only resolve conflicts automatically with arbitrary policies, such as "last writer wins."

In the context of feature modeling, we argue that collaborators should participate in the conflict resolution to preserve the intentions of all conflicting operations. To this end, we use a *multi-versioning concurrency-control technique* (Chen 2001a; Stefik et al. 1987; Wulf 1995). In contrast to the single-document 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). Multi-versioning techniques can be implemented in two ways, mainly differentiating whether they expose only a single document version (and keep others only internally) or expose multiple, conflicting document versions (Sun et al. 2002, 2004). The first type of techniques has the problem of committing to one particular version based on arbitrary rules. Moreover, feature modeling poses semantic problems during conflict resolution that are hard to address in an automated process. The second type, called Multi-Version Multi-Display (MVMD) (Sun and Chen 2002; Chen and Sun 2001b), lets users decide which new document version should be used in case of a conflict. In Fig. 3, we show how this technique allocates two conflicting update operations to two different versions of an edited feature model, 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 feature model. As MVMD fulfills all of our requirements, we use it as basis for our collaborative feature-model editor. However, it needs further adjustments to support feature modeling, such as defining a conflict relation for feature-modeling operations, which we describe in Section 3.2.

MVMD has been introduced in the Graphics Collaborative Editing (GRACE) system to maintain multiple versions of edited objects in the face of conflicts (Sun and Chen 2000; 2002; Chen and Sun 2001b). It groups operations according to whether they are *conflicting*

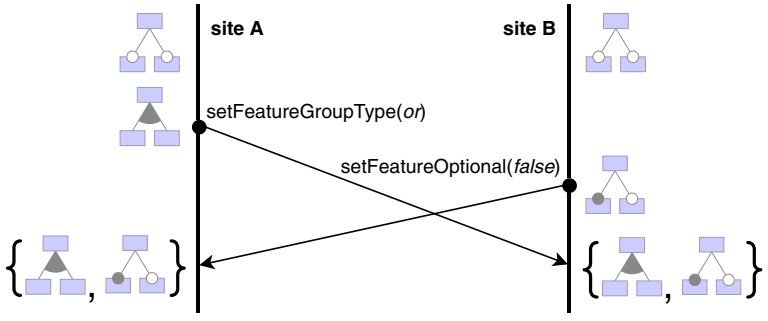


Fig. 3 A conflict scenario detected with the multi-version multi-display technique. Both sites arrive at the same maximum compatible group set (cf. Definition 9)

or *compatible*. To determine algorithmically whether two operations are in conflict, we utilize a *conflict relation* (Sun and Chen 2002; Xue et al. 2003):

Definition 8 (Conflict Relation) 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, namely $O_a \not\otimes O_b$, they are *compatible*, denoted as $O_a \odot O_b$. Only concurrent operations may conflict, that is, for any operations O_a and O_b, $O_a \nparallel O_b \Rightarrow O_a \odot O_b$.

The specific conflict relation used in GRACE states that two concurrent operations are in conflict if they set the same attribute of the same object to different values. However, the conflict relation of GRACE is not suitable for collaborative, real-time feature modeling, as it only captures *overwrite* conflicts, while feature modeling can cause other kinds of conflicts as well. For example, the *acyclic* property in Definition 3 can be violated by concurrently moving two features below one another. In addition, the conflict relation of GRACE cannot capture semantic inconsistencies, such as *dead features* or *redundant constraints* (Benavides et al. 2010; Felfernig et al. 2013; Kowal et al. 2016). Therefore, we introduce new conflict relations specific to feature modeling in Section 3.2.

By using a proper conflict relation, we can create different versions of a feature model whenever a conflict occurs. However, as we consider multiple remotely connected collaborators, we may also need to resolve conflicts for three or more concurrent operations at a time. In such a situation, if we would create a different version for each conflicting operation, the conflict resolution could overwhelm the collaborators. To avoid this problem, we rely on the *combined effect* of the MVMD technique to minimize the number of versions, while still preserving the intentions of all operations (Sun and Chen 2002):

Definition 9 (Combined Effect) Let $GO = \{O_1, O_2, \ldots, O_n\}$ be a group of operations targeting the same object.

- A *compatible group* CG is a subset $CG \subseteq GO$ where $\forall O_a, O_b \in CG\colon O_a \odot O_b$.
- A compatible group CG is *maximum* if $\forall O_a \in GO\colon (O_a \in CG \vee \exists O_b \in CG\colon O_a \otimes O_b)$.
- A *maximum compatible group set* $MCGS = \{CG_1, CG_2, \ldots, CG_m\}$ is a set of maximum compatible groups where all of the following conditions are true:
  - $\forall O \in GO\colon \exists CG_i \in MCGS\colon O \in CG_i$
  - $\forall O_a, O_b \in GO\colon (O_a \odot O_b \Rightarrow \exists CG_i \in MCGS\colon O_a, O_b \in CG_i)$
  - all maximum compatible groups belonging to GO are included in $MCGS$.
- Given a maximum compatible group set $MCGS$, the *combined effect* for GO is that
  - for each $CG \in MCGS$, one version V_{CG} of the targeted object is created, and
  - for each $CG \in MCGS$, all included operations are applied to V_{CG}.

By using an MCGS, we can generate a minimum number of versions, while still satisfying all submitted operations' intentions. It can be shown that the MCGS, and thus the combined effect for a given group of operations, is unique (Sun and Chen 2002). For example, given a group of operations $\{O_1, O_2, O_3, O_4\}$ and the conflict relation $O_1 \odot O_2$, $O_1 \odot O_3$, $O_1 \odot O_4$, $O_2 \otimes O_3$, $O_2 \odot O_4$, and $O_3 \otimes O_4$, their unique MCGS is $\{\{O_1, O_3\}, \{O_1, O_2, O_4\}\}$. The MCGS for a group of operations can be constructed incrementally at any site using the MOVIC algorithm proposed by Sun and Chen (2002). When targeting a single object, this algorithm has been proven to converge to the same MCGS at all sites (Sun and Chen 2002).

3 Designing a Collaborative, Real-Time Feature Model Editor

In this section, we describe the concepts of our technique that enables users to collaboratively edit the same feature model. For this purpose, we introduce an operation model, conflict detection, and conflict resolution.

3.1 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 these, which makes consistency checking, resolving conflicts, and reasoning about the editor's correctness more complex. To address this issue, we use a two-layered operation architecture (Sun et al. 2006), 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 a feature model and are suitable to use in concurrency-control techniques. COs then expose actual feature-modeling operations to the application. A COs is a sequence of POs that are executed in a defined order, and thus can express all feature-modeling operations mentioned in Section 2.2.

Using this two-layer architecture instead of single set of operations has several advantages: When detecting conflicts between operations, we can focus on POs and do not need to analyze COs, since they are PO sequences. For the same reason, we can implement multi-target operations, such as *remove*, by composing several single-target COs into a new CO. Also, to extend the editor with additional operations, we need to implement only new COs, without making major changes to the conflict detection. In the following, we provide an overview of the current set of operations that we support.

3.1.1 Primitive Operations

Single-user feature-modeling tools implement COs to create, remove, and modify features as well as cross-tree constraints in various ways. We define five POs that serve as building blocks for such COs. For each PO, we specify pre- and postconditions with formal semantics, 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. For the sake of brevity, we only present the formal details of the primitive operation **createFeaturePO** and provide the details of all other POs in Appendix A.1.

The operation **createFeaturePO** (PO 1) creates a new feature in the feature model:

PO 1: **createFeaturePO**(F^{ID})

Creates a feature with a new globally unique identifier. Assigns default values to the feature's attributes. The new feature is initially graveyarded to facilitate conflict detection (to insert it into the feature tree, a subsequent update PO can be used).

PRECONDITIONS

$F^{ID} \in \boldsymbol{ID}$

$F^{ID} \notin \boldsymbol{F}^{ID}_{FM}$

POSTCONDITIONS

$F^{ID} \in \boldsymbol{F}^{ID}_{FM'}$

$FM'.\boldsymbol{F}(F^{ID}).parentID = \dagger$

$FM'.\boldsymbol{F}(F^{ID}).optional = true$

$FM'.\boldsymbol{F}(F^{ID}).groupType = and$

$FM'.\boldsymbol{F}(F^{ID}).abstract = false$

$FM'.\boldsymbol{F}(F^{ID}).name =$ `New Feature`

An existing feature can be updated using the operation **updateFeaturePO** (PO 2). This operation allows to modify any attribute of any existing feature. For creating and updating cross-tree constraints, we define the analogous operations **createConstraintPO** (PO 3) and **updateConstraintPO** (PO 4). We do not require any operations for removing features or cross-tree constraints, because they can be expressed as *update* operations, setting the *parentID* to † (i.e., graveyarding). Finally, we define a single *assertion* operation, **assertNoChildAddedPO** (PO 5), which we need to detect conflicts related to operations that graveyard features (cf. Section 3.2). With these five POs, we can assemble more complex COs for feature-modeling.

3.1.2 Compound Operations

To allow for high-level modeling operations, we employ COs. Each CO consists of a sequence of POs, which are applied atomically. A CO is defined by its associated preconditions and an algorithm that generates the CO's PO sequence, which has to ensure the preconditions of each comprised PO. To model control flow and loops when generating COs, we use regular `if` and `for` statements; however, their application is strictly sequential to facilitate conflict detection. Whenever a user requests to generate a CO, its preconditions are checked against the current feature model *FM*, then the CO's algorithm is invoked with the required arguments, such as the feature model (*FM*), a feature parent (*FP*), or feature children (*FC*). The generated CO is first applied locally and then propagated to other users.

In the following, we present the COs that we implemented in our tool. We introduce six feature-related operations for inserting and arranging features in the feature tree and three constraint-related operations for creating, modifying, and removing constraints. However, to allow for new functionality, more COs can be designed on top of the existing POs. Analogous to POs, we present only one CO and describe all other COs in Appendix A.2.

The operation **createFeatureBelow** (CO 1) creates and inserts a feature in the feature tree as the child of an existing feature:

CO 1: **createFeatureBelow**(FM, F^{ID}, FP^{ID})

Creates a feature F^{ID} below another feature FP^{ID}.
Require: $F^{ID} \in \boldsymbol{ID}$, $F^{ID} \notin \boldsymbol{F}^{ID}_{FM}$, $FP^{ID} \in \boldsymbol{F}^{ID}_{FM}$

```
1: function CREATEFEATUREBELOW(FM, F^ID, FP^ID)
2:     return [createFeaturePO(F^ID),                    ▷ create the new feature
3:             updateFeaturePO(F^ID, parentID, †, FP^ID)] ▷ insert it into the tree
4: end function
```

Similarly, the operation **createFeatureAbove** (CO 2) creates a new feature and inserts it in the feature tree as parent of an existing feature. If the existing feature was not the root feature, its original parent becomes the parent of the new feature. Features can also be moved within the feature tree using the operation **moveFeatureSubtree** (CO 3), which changes the parent of a subtree to another existing feature. With the operation **removeFeatureSubtree** (CO 4), an entire subtree can be removed from the feature tree by graveyarding it. In contrast, the operation **removeFeature** (CO 5) graveyards only a single feature and pulls its children up by replacing the children's parent with its own parent. For modifying additional feature attributes, such as the *optional* flag, *group type*, and *feature name*, we introduce operations to set the values of these attributes. For instance, the operation **setFeatureOptional**

(CO 6) sets the attribute *optional* of a single feature to either *true* or *false*. The definitions of *setFeatureGroupType*, *setFeatureAbstract*, and *setFeatureName* are analogous.

The operation **createConstraint** (CO 7) creates a new constraint and adds it to the feature model. An existing constraint can be modified by applying the operation **setConstraint** (CO 8), which replaces a given constraint with a new one. With the operation **removeConstraint** (CO 9), an existing constraint can be graveyarded, and thus effectively removed from the feature model.

3.1.3 Applying Operations

As POs and COs are only descriptions of edits on a feature model, we further need to define how to apply them to produce a new (modified) feature model. We define application functions for POs and COs as follows:

Definition 10 Let $FM \in \boldsymbol{FM}$. 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*.

Our tool provides *applyPO* and ensures all postconditions of POs. Note that *applyCO* does not treat any CO specially, which facilitates conflict detection and extensions. From the definitions of the application function *applyCO* and the COs, we can derive that *applyCO* always preserves the syntactical correctness of feature models (cf. Section 2.1) when no concurrent operations are submitted (i.e., in a single-user scenario):

Theorem 1 *Let* $FM \in \boldsymbol{FM}$ *be a legal feature model. Further, let CO be a compound operation whose preconditions are satisfied with regard to FM. Then,* $FM' = applyCO(FM, CO) \in \boldsymbol{FM}$*, that is, FM' is again a legal feature model.*

We omit the proof for this theorem here, but provide it in Appendix A.3.

3.2 Conflict Detection

For any collaborative work, it is essential to detect conflicts between the edits users perform (e.g., changing the name of the same feature to different values). In the following, we describe how we extended the MVMD technique with new 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. The globally targeted object strategy, causal directed acyclic graph, outer conflict relation, and topological sorting strategy are novel extensions to the MVMD technique. These extensions are, in principle, agnostic to feature modeling and may be adapted to other domains of collaborative editing in the future. Building on these extensions, we then discuss an inner conflict relation that is specific to feature modeling.

3.2.1 Globally Targeted Object Strategy

The combined effect specified in Definition 9 considers *targeted objects* for which multiple versions may be created. In the graphics-editing system GRACE, an *object* refers to a distinct graphical element on the screen, which may be duplicated if a conflict appears (as we

show in Fig. 3). However, we argue that in a collaborative feature-model editor, only one *globally targeted object*, namely the feature model, should be considered. Feature models impose additional semantics (i.e., the variability modeled in an SPL), which can be conflicted simply by having two versions of the same feature. Further, certain conflicts, such as cycle-introducing *moveFeatureSubtree* operations, cannot be resolved when features are considered as individual objects. Considering the feature model as one global object enables more comprehensive conflict detection and avoids the introduction of semantic inconsistencies. So, conflicts cause multiple versions of the feature model, which collaborators can then inspect to determine the desired resolution.

3.2.2 Causal Directed Acyclic Graph

In Section 2.2, we introduced a causal ordering for tracking operations' concurrency relationships in the tool. However, the conflict relations for collaborative feature modeling 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 (Mattern 1988; Schwarz and Mattern 1994). 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 11 (Causal Directed Acyclic Graph) 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 $E = \{(O_a, O_b) \mid O_a, O_b \in GO \wedge O_a \rightarrow O_b\}$ is the set of edges. Then, the set of *causally preceding operations* for an $O \in GO$ is defined as $CP_G(O) \coloneqq \{O_a \mid (O_a, O) \in E\}$.
- 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 $CIP_G(O) \coloneqq \{O_a \mid (O_a, O) \in E'\}$.

The transitive reduction of a graph removes all edges that represent only transitive dependencies (Aho et al. 1972). Therefore, an operation O_a causally *immediately* precedes another operation O_b when there is no operation O_x such that $O_a \rightarrow O_x \rightarrow O_b$. Both, the CDAG and its transitive reduction are unique, so that the CP and CIP sets for an operation are well-defined. Each collaborating site has a copy of the current CDAG, which is incrementally constructed and includes all previously generated and received operations.

3.2.3 Outer Conflict Relation

The GRACE system uses solely the operation 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 operation metadata, but also the targeted feature model. This is possible due to our globally targeted object strategy. Such a conflict relation may inspect the involved operations and apply them to the targeted feature model to check whether their application introduces any inconsistencies.

In order for such a conflict relation to work properly, we need to address the technical difficulty of finding the appropriate targeted feature model that the two operations should be applied on, which may differ from the currently displayed one. Such a feature model can be derived from the checked operations' common ancestor in the CDAG and their causally

preceding operations. However, this feature model is only meaningfully defined if all causally preceding operations of both checked operations are compatible. Otherwise, the intention preservation property may be violated, so that the conflict relation would rely on potentially inconsistent and unexpected feature models. Thus, an appropriate *outer conflict relation*, termed $\otimes_O$, must propagate any conflict between two operations to all causally succeeding operations. We can express this property as follows:

Definition 12 (Conflict Propagation) Let GO be a group of operations and $O_a, O_b \in GO$. Then, $O_a \otimes_O O_b$ if at least one of the following conditions is true:

- $\exists O_x \in GO\colon O_x \rightarrow O_a \wedge O_x \otimes_O O_b$,
- $\exists O_y \in GO\colon O_y \rightarrow O_b \wedge O_a \otimes_O O_y$, or
- $\exists O_x, O_y \in GO\colon O_x \rightarrow O_a \wedge O_y \rightarrow O_b \wedge O_x \otimes_O O_y$.

To determine the outer conflict relation for two given COs, we introduce OUTERCONFLICTING (cf. 1), a recursive algorithm that uses the CDAG to propagate conflicts as defined above:

Algorithm 1 Computing the conflict-propagating outer conflict relation $\otimes_O$.

Require: G is the CDAG for a group of operations GO,
$CO_a, CO_b \in GO$

```
1: function OUTERCONFLICTING(G, CO_a, CO_b)
2:     if CO_a ∦ CO_b ∨ CO_a = CO_b then return false
3:     if ∃CIPO_a ∈ CIP_G(CO_a), CIPO_b ∈ CIP_G(CO_b):
4:             OUTERCONFLICTING(G, CIPO_a, CIPO_b)
5:         ∨ ∃CIPO_b ∈ CIP_G(CO_b): OUTERCONFLICTING(G, CO_a, CIPO_b)
6:         ∨ ∃CIPO_a ∈ CIP_G(CO_a): OUTERCONFLICTING(G, CIPO_a, CO_b)
7:     then return true
8:     return CO_a ⊗_I CO_b
9: end function
```

OUTERCONFLICTING ensures that there is a well-defined feature model for subsequent conflict detection, which enables us to check arbitrary consistency properties—with the disadvantage that in few cases operations may be falsely flagged as conflicting, because we make no distinction between time and causality. Essentially, OUTERCONFLICTING defers the conflict detection to the inner conflict relation $\otimes_I$, which we discuss below. So, we separate two concerns, that is, the technical problem of ensuring an appropriate targeted feature model ($\otimes_O$, instead of $\otimes$ in Definition 9), as well as conflict-detection rules specific to collaborative feature modeling ($\otimes_I$). Using OUTERCONFLICTING, we can compute $\otimes_O$ as follows:

Definition 13 (Outer Conflict Relation) 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. Otherwise, they are *outer compatible*, i.e., $CO_a \odot_O CO_b$.

3.2.4 Topological Sorting Strategy

The outer conflict relation $\otimes_O$ ensures that we can produce an appropriate feature model. To actually produce such a feature model, we employ a *topological sorting* of operations

according to their causality relationships, which we implemented as APPLYCOMPATIBLEGROUP (Algorithm 2). We need APPLYCOMPATIBLEGROUP to apply (unordered) sets of mutually compatible operations, namely CGs, to a feature model. Because the application order of operations is important for producing a correct result, our topological sorting strategy ensures an application order that respects all causal relationships captured in the CDAG. Note that a topological sorting exists and can be constructed for any CDAG, but is not necessarily unique, since concurrent operations may still be swapped (Kahn 1962). This, however, does not impact the correctness of our technique, since concurrent compatible operations always commute in our tool, as guaranteed by the *no-overwrites* rule that we introduce next.

Algorithm 2 Applying a set of mutually compatible operations to a feature model.

Require: G is the CDAG for a group of operations GO,
$FM \in \boldsymbol{FM} \wedge CG$ is a compatible group for GO
1: **function** APPLYCOMPATIBLEGROUP(G, FM, CG)
2: $COs \leftarrow$ topological sorting of CG according to G
3: **while** $CO \leftarrow COs.next()$ **do** $FM \leftarrow applyCO(FM, CO)$
4: **return** FM
5: **end function**

3.2.5 Inner Conflict Relation

The *inner conflict relation* $\otimes_I$ detects conflicts that are specific to feature modeling. To this end, we first introduce SYNTACTICALLYCONFLICTING (Algorithm 3), which determines whether two compound operations have a *syntactic conflict* that concerns basic syntactic properties of feature models.

First, SYNTACTICALLYCONFLICTING applies one compound operation, CO_y, to an appropriate feature model. We produce this feature model with APPLYCOMPATIBLEGROUP from the initial feature model, to which no operations from the CDAGs have been applied, yet. Then, the POs that CO_x comprises are subsequently applied to this feature model, so that, finally, CO_x and CO_y are both applied. While applying these COs, we can inspect the current feature model for potential consistency problems using the following set of *conflict-detection rules*.

The *no-overwrites* rules state that no two concurrent operations may update the same feature's or cross-tree constraint's attribute to a new value. Thus, the intentions of concurrent updates on the same model element are preserved by accommodating them into different versions. Next, the *no-cycles* rule preserves the preconditions of the *moveFeatureSubtree* CO by forbidding to move (i.e., updating the *parentID*) features that are themselves moved. So, we preserve the *acyclic* property from Definition 3. The *no-graveyarded* rules guarantee that no operation targets a graveyarded feature or cross-tree constraint. This way, no remove operation can override other collaborators' update operations. Further, the *no-group-optional* rule detects when a *groupType* update would override a concurrent update of an *optional* attribute and ensures that the root feature is always mandatory. Finally, the *assert-no-child-added* rule ensures an assumption of the *removeFeature* CO, namely that removed features' children are not changed concurrently. This helps to preserve the *single root* property of legal feature models (cf. Definition 3).

Algorithm 3 Stepwise computation of syntactic conflicts between compound operations.

Require: G is the CDAG for a group of operations GO,
$FM_{init} \in \boldsymbol{FM}$ is the initial feature model for G,
$CO_x, CO_y \in GO$

function SYNTACTICALLYCONFLICTING(G, FM_{init}, CO_x, CO_y)
  $FM_{precCO_y} \leftarrow$ APPLYCOMPATIBLEGROUP($G, FM_{init}, CP_G(CO_y)$)
  $FM_{CO_y} \leftarrow$ APPLYCO(FM_{precCO_y}, CO_y)
  $FM \leftarrow$ APPLYCOMPATIBLEGROUP($G, FM_{CO_y}, CP_G(CO_x) \setminus CP_G(CO_y)$)
  while $CO_x.hasNext()$ **do** ▷ subsequently apply POs in CO_x in order
    $PO_x \leftarrow CO_x.next()$
    if
      ▷ **No-Overwrites Rule** for features
      $PO_x = updateFeaturePO(F^{ID}, attribute, oldValue, newValue)$
        $\wedge\ oldValue \neq FM.\boldsymbol{F}(F^{ID}).[attribute]$
      ▷ **No-Overwrites Rule** for cross-tree constraints
      $\vee\ PO_x = updateConstraintPO(C^{ID}, attribute, oldValue, newValue)$
        $\wedge\ oldValue \neq FM.\boldsymbol{C}(C^{ID}).[attribute]$
      ▷ **No-Cycles Rule**
      $\vee\ PO_x = updateFeaturePO(F^{ID}, attribute, oldValue, newValue)$
        $\wedge\ attribute = parentID$
        $\wedge\ \{F^{ID} \mid newValue \preceq_{FM_{CO_y}} F^{ID}\} \neq \{F^{ID} \mid newValue \preceq_{FM_{precCO_y}} F^{ID}\}$
      ▷ **No-Graveyarded Rule** for targeted features
      $\vee\ PO_x = updateFeaturePO(F^{ID}, attribute, oldValue, newValue)$
        $\wedge\ F^{ID} \in \boldsymbol{F}^{\dagger}_{FM}$
        $\wedge\ \neg(attribute = parentID \wedge oldValue = \dagger)$
      ▷ **No-Graveyarded Rule** for parent features
      $\vee\ PO_x = updateFeaturePO(F^{ID}, attribute, oldValue, newValue)$
        $\wedge\ attribute = parentID$
        $\wedge\ newValue \in \boldsymbol{F}^{\dagger}_{FM}$
      ▷ **No-Graveyarded Rule** for cross-tree constraints
      $\vee\ PO_x = updateConstraintPO(C^{ID}, attribute, oldValue, newValue)$
        $\wedge\ C^{ID} \in \boldsymbol{C}^{\dagger}_{FM}$
        $\wedge\ \neg(attribute = \dagger \wedge oldValue = true)$
      ▷ **No-Group-Optional Rule**
      $\vee\ PO_x = updateFeaturePO(F^{ID}, attribute, oldValue, newValue)$
        $\wedge\ attribute = optional$
        $\wedge\ (FM.\boldsymbol{F}(F^{ID}).parentID = \bot$
          $\vee\ FM.\boldsymbol{F}(FM.\boldsymbol{F}(F^{ID}).parentID).groupType \neq and)$
      ▷ **Assert-No-Child-Added Rule**
      $\vee\ PO_x = assertNoChildAddedPO(F^{ID})$
        $\wedge\ \{FC^{ID} \in \boldsymbol{F}^{ID}_{FM_{CO_y}} \mid FM_{CO_y}.\boldsymbol{F}(FC^{ID}).parentID = F^{ID}\} \setminus$
          $\{FC^{ID} \in \boldsymbol{F}^{ID}_{FM_{precCO_y}} \mid FM_{precCO_y}.\boldsymbol{F}(FC^{ID}).parentID = F^{ID}\} \neq \varnothing$
      then return *true*
    $FM \leftarrow$ APPLYPO(FM, PO_x)
  end while
  return *false*
end function

Building on SYNTACTICALLYCONFLICTING, we introduce INNERCONFLICTING (Algorithm 4) to determine $\otimes_I$ for two given compound operations.

Algorithm 4 Computing the feature-modeling-aware inner conflict relation $\otimes_I$.

Require: G is the CDAG for a group of operations GO,
$FM_{init} \in \boldsymbol{FM}$ is the initial feature model for G,
$CO_a, CO_b \in GO$

1: **function** INNERCONFLICTING(G, FM_{init}, CO_a, CO_b)
2: **if** $CO_a \nparallel CO_b \vee CO_a = CO_b$ **then return** *false*
3: **if** SYNTACTICALLYCONFLICTING(G, FM_{init}, CO_a, CO_b) **then return** *true*
4: **if** SYNTACTICALLYCONFLICTING(G, FM_{init}, CO_b, CO_a) **then return** *true*
▷ check additional semantic properties
5: $FM \leftarrow$ APPLYCOMPATIBLEGROUP($G, FM_{init}, CP_G(CO_a) \cup CP_G(CO_b) \cup \{CO_a, CO_b\}$)
6: **if** $\exists SP \in \boldsymbol{SP}$: SP($FM$) $=$ *true* **then return** *true*
7: **return** *false*
8: **end function**

To ensure the symmetry of $\otimes_I$, INNERCONFLICTING uses SYNTACTICALLYCONFLICTING to check for syntactic conflicts in both directions. Finally, INNERCONFLICTING may check additional arbitrary *semantic properties* of a feature model that includes the effects of CO_a and CO_b. A semantic property $SP \in \boldsymbol{SP}$ is a deterministic function $SP\colon \boldsymbol{FM} \rightarrow \{true, false\}$ that returns whether a given legal feature model includes a semantic inconsistency. $\boldsymbol{SP}$ may be adjusted according to the collaborators' needs when the tool is initialized. For instance, collaborators may want to ensure that the modeled SPL has at least one product (*void feature model* analysis) at all times and does not include *dead features*, *false-optional features*, or any *redundant cross-tree constraints* (Benavides et al. 2010; Felfernig et al. 2013; Kowal et al. 2016). Note that our technique allows only pairwise conflict detection of operations, since interactions of higher order are hard to detect (Bowen et al. 1989; Calder et al. 2003; Apel et al. 2013b). Using INNERCONFLICTING, we can compute $\otimes_I$ as follows:

Definition 14 (Inner Conflict Relation) Two compound operations CO_a and CO_b are in *inner conflict*, i.e., $CO_a \otimes_I CO_b$, iff INNERCONFLICTING(G, FM_{init}, CO_a, CO_b) $= true$,

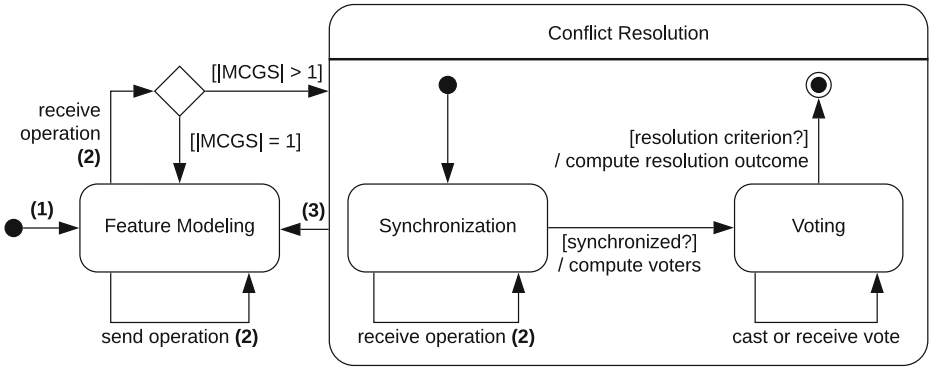


Fig. 4 UML state diagram of feature modeling at a single site with our tool. The substate describes our conflict resolution process

where G and $FM_{init} \in \boldsymbol{FM}$ are the current CDAG and initial feature model at the site that executes INNERCONFLICTING. Otherwise, they are *inner compatible*, i.e., $CO_a \odot_I CO_b$.

3.2.6 Feature-Modeling Loop

In the previous sections, we described algorithms for calculating two conflict relations for collaborative feature modeling. We apply these algorithms in a feature-modeling loop, which we show in Fig. 4. In this loop, a site may, after initialization **(1)**, send (i.e., generate and propagate) new feature-modeling operations **(2)**. Moreover, a site receives all operations generated at other sites **(2)**. Receiving an operation may lead to a conflict, which triggers our conflict resolution mechanism (detailed in Section 3.3). We proceed to describe site initialization and operation processing in more detail.

Algorithm 5 Initializing a newly joined site.

Require: $FM_{init} \in \boldsymbol{FM}$ is the initial feature model
1: **function** INITIALIZESITE(FM_{init})
2: $G \leftarrow (\varnothing, \varnothing)$
3: $MCGS \leftarrow \varnothing$
4: $uiState \leftarrow FM$
5: **return** $(G, MCGS, FM_{init}, uiState)$
6: **end function**

When a new site (i.e., collaborator) joins an editing session **(1)**, it must call INITIALIZESITE (Algorithm 5) with the current feature model, which is provided by another site. This algorithm returns the site's initial state, which consists of an empty CDAG, empty MCGS, and the initial feature model. In addition, the returned *uiState* is passed to the user interface of the feature model editor, to be rendered as a feature diagram.

Algorithm 6 Processing a newly sent or received operation.

Require: G is the CDAG for a group of operations GO,
$MCGS$ is the maximum compatible group set for GO,
$FM_{init} \in \boldsymbol{FM}$ is the initial feature model for G,
$CO \notin GO$
1: **function** PROCESSOPERATION($G, MCGS, FM_{init}, CO$)
2: $GO' \leftarrow GO \cup \{CO\}$
3: $G' \leftarrow$ the causal directed acyclic graph for GO'
4: $MCGS' \leftarrow$ MOVIC($G, MCGS, FM_{init}, CO$)
5: **if** $MCGS' = \varnothing$ **then** $uiState \leftarrow FM_{init}$
6: **else if** $MCGS' = \{CG\}$ **then** $uiState \leftarrow$ APPLYCOMPATIBLEGROUP(G, FM_{init}, CG)
7: **else** $uiState \leftarrow$ conflict descriptor for $MCGS'$
8: **return** $(G', MCGS', FM_{init}, uiState)$
9: **end function**

Once initialized, the site may send and receive operations **(2)**, both of which are processed by the same algorithm, PROCESSOPERATION (Algorithm 6). To this end, the site calls PROCESSOPERATION with its current state (CDAG, MCGS, and initial feature model) and the newly received or sent CO. PROCESSOPERATION adds the new operation to the

CDAG and detects potential conflicts with MOVIC (the MCGS construction algorithm proposed by Sun and Chen 2002), which utilizes our outer conflict ($\otimes_O$) and compatibility ($\odot_O$) relations. Depending on the constructed MCGS, the new *uiState* is constructed: If $MCGS'$ is empty (just after site initialization) or consists of one version (i.e., compatible group), a feature model can be constructed with APPLYCOMPATIBLEGROUP and passed to the user interface for rendering. In case of a conflict (i.e., two or more compatible groups), a conflict descriptor that comprises detailed information about the conflicting operations and involved collaborators is passed to the editor to aid in the conflict resolution (cf. Section 3.3). Note that, in the case of generating a new operation, MOVIC is guaranteed to not detect any conflicts, because a locally generated operation always causally succeeds all previously seen operations. In particular, the resulting feature diagram can be rendered immediately, without waiting for other sites to acknowledge the change, which makes our technique optimistic (in contrast to, for example, locking).

3.3 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 4) during which collaborators examine alternative feature-model versions and negotiate a specific version (Stefik et al. 1987; Wulf 1995). To this end, we allow collaborators to cast *votes* for their preferred feature model versions, which allows for fair and flexible conflict resolution (Dennis et al. 1998; Gibbs 1989; Morris et al. 2004).

In our process, a site forbids any further editing (referred to as *freeze*) when a conflict is detected (i.e., $|MCGS| > 1$). This forces collaborators to address the conflict, avoiding any further divergence. Freezing the tool also ensures the correctness of our technique, as the MVMD technique has only been proven correct for this use case (Sun and Chen 2002; Xue et al. 2003). After freezing, the tool synchronizes all sites (i.e., waits until all sites received all pending operations) so that all collaborators are aware of all versions before starting the voting process, which is the only synchronization period that is required by our technique. Next, a set of voters V (i.e., collaborators that are eligible to vote) may be flexibly computed based on the collaborators' preferences. For example, we implemented the following possibilities:

- $V = \varnothing$, that is, no collaborator may vote.
- V consists of all collaborators that are involved in the conflict (they are *involved* in a conflict when they have submitted an operation O such that $\exists CG \in MCGS\colon O \notin CG$).
- V contains all collaborators participating in the editing session.

To start the voting, we initialize a set of vote results, VR, 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. After a vote is processed at a site, a *resolution criterion* decides whether the voting phase is complete. Again, we implemented several resolution criteria in our prototype, including:

- $|VR| = 0$, that is, the voting phase is concluded immediately, which is useful to implement a *reject-all-conflicts* policy.
- $|VR| > 0$, so the voting phase is concluded after the first vote has been cast, which roughly corresponds to a *first-writer-wins* policy.
- $|VR| = |V|$, that is, all collaborators have to vote for the voting phase to conclude.

– $|VR| = |V|$ or there are two distinct collaborators with different votes. This way, the voting phase will conclude after all collaborators have cast a vote or there is any dissent among collaborators.

When the chosen resolution criterion has been triggered, we compute the elected feature-model version (the *resolution outcome*) from *VR*. To be able to discard all conflicting changes (e.g., when collaborators cannot agree on one version), we introduce the *neutral compatible group*, which can be calculated just before the voting phase starts:

Definition 15 (Neutral Compatible Group) Let *MCGS* be a maximum compatible group set. Then, the *neutral compatible group* NCG_{MCGS} for *MCGS* is defined as $NCG_{MCGS} \coloneqq \bigcap_{CG \in MCGS} CG$.

In particular, the neutral CG contains all operations that are applied in every feature-model version and have no conflict potential with any other operations. With this in mind, we implemented different resolution outcomes in our tool:

– NCG_{MCGS} wins unconditionally, undoing any conflicting changes in the process.
– *Plurality*: The $CG \in MCGS$ with the most votes in *VR* wins. In case of a tie or $|VR| = 0$, NCG_{MCGS} wins.
– *Majority*: The $CG \in MCGS$ with the most votes in *VR* wins if it has more than $|VR|/2$ votes, otherwise NCG_{MCGS} wins.
– *Consensus*: If $|VR| > 0$ and all votes in *VR* agree on a single $CG \in MCGS$, *CG* wins, otherwise NCG_{MCGS} wins.

When the outcome has been computed at a site, our tool calls RESOLVECONFLICT (Algorithm 7) with that outcome, that is, the chosen compatible group *CG*. RESOLVECONFLICT then transitions back to the feature modeling phase **(3)** by re-initializing the site with the feature model corresponding to *CG* and resetting the CDAG and MCGS in the process. Then, the collaborators can proceed with modeling activities until the next conflict is detected.

Algorithm 7 Resolving a conflict by re-initializing the site with a new feature model.

Require: G is the CDAG for a group of operations GO,
$MCGS$ is the maximum compatible group set for GO,
$FM_{init} \in \boldsymbol{FM}$ is the initial feature model for G,
$CG \in MCGS \cup \{NCG_{MCGS}\}$

1: **function** RESOLVECONFLICT(G, $MCGS$, FM_{init}, CG)
2: **return** INITIALIZESITE(APPLYCOMPATIBLEGROUP(G, FM_{init}, CG))
3: **end function**

4 Correctness

A particular challenge in designing optimistic schemes for concurrency control is to prove their correctness, which is essential to avoid defective systems (Imine et al. 2006; Randolph et al. 2013; Sun et al. 2014; Cormack 1995). To address this challenge, we chose the MVMD technique, which has been carefully designed with the CCI model in mind (cf. Definition 7). So, we have to reason solely about the correctness of our adaptations for collaborative

feature modeling, the CDAG, and outer as well as inner conflict relations. By ensuring compliance with the CCI model, we avoid inconsistent feature models, raise the collaborators' confidence in the system, and thus allow effective and efficient editing of feature models.

4.1 Causality Preservation

A collaborative, real-time editing system preserves causality when it ensures that for any operation, all causally preceding operations are processed beforehand (Sun et al. 1998). This is satisfied in client-server architectures and can be achieved in peer-to-peer architectures with existing techniques that are not specific to feature modeling (Fidge 1988; Sun et al. 1998, 1999). In client-server architectures with well-ordered message channels, causality preservation is achieved trivially (Sun and Sosič 1999). For other topologies (e.g., peer-to-peer architectures), suitable causality preservation schemes have been proposed in the literature (Fidge 1988; Sun et al. 1998). These schemes are not specific to feature modeling and can be applied to our concept without modifications. Assuming a suitable causality preservation scheme, we can show that at any site, an operation's set of causally (immediately) preceding operations is fully known when the operation is being processed.

Theorem 2 *Let G be the CDAG for a group of operations GO and $O \in GO$. Further, let G' be the current CDAG at any site at any time. Then, if G' includes O, also $CP_{G'}(O) = CP_G(O)$, that is, the set of causally preceding operations is computed correctly at said site.*

Proof Recall that G' is incrementally constructed as a site processes operations. That is, when an operation is sent or received, it is inserted into G' by PROCESSOPERATION. As guaranteed by a suitable causality preservation scheme, any $O_x \in CP_G(O)$ is processed before O at all sites. Thus, when O is processed, G' already includes all operations in $CP_G(O)$ and $CP_G p(O) = CP_G(O)$. □

Below, we use this property to reason about the outer and inner conflict relation and the combined effect of a maximum compatible group set.

4.2 Convergence

A collaborative, real-time editing system converges if all sites arrive at identical feature models after a group of operations has been processed, although the arrival order may differ from site to site (Sun et al. 1998). To this end, we rely on the correctness of the MVMD technique (and the MOVIC algorithm in particular), as described by Sun and Chen (2002). Sun and Chen prove that MOVIC converges for operations that target the same version of an object. We enforce this with our globally targeted object strategy (cf. Section 3.2) and by freezing all sites when a conflict occurs (cf. Section 3.3). However, we introduced two new conflict relations and a topological sorting strategy in Section 3.2, which require further verification to confirm that our system converges.

First, we examine APPLYCOMPATIBLEGROUP, which applies a set of mutually compatible operations (a compatible group) to a feature model by means of a topological sorting. We mentioned that multiple topological sortings may exist for a compatible group, which only differ in the execution order of concurrent operations. As APPLYCOMPATIBLEGROUP requires the supplied operations to be mutually compatible, we only have to show that any two compatible concurrent operations commute in our system. This follows from the

no-overwrites rules introduced in SYNTACTICALLYCONFLICTING: COs that modify the same feature or cross-tree constraint's attribute are considered conflicting. Such operations, however, are the only operations in our system that do not necessarily commute. Thus, compatible concurrent operations always commute in our system. No matter which topological sorting APPLYCOMPATIBLEGROUP determines, it computes the same feature model for a given compatible group. In particular, it is also *stable across sites*, that is, the computed result does not depend on the executing site. We proceed to prove the precondition that APPLYCOMPATIBLEGROUP is only invoked with sets of mutually compatible operations:

- In SYNTACTICALLYCONFLICTING, APPLYCOMPATIBLEGROUP prepares feature models by applying subsets of operations' causally preceding operation sets (their *CP sets*). Note that the CP set of an operation O_a only includes mutually compatible operations: If there were $O_b, O_c \in CP_G(O_a)$ such that $O_b \otimes_O O_c$, the system would have been frozen until the conflict is resolved, and O_a would have never causally succeeded both, O_b and O_c. Thus, the CP subsets used in SYNTACTICALLYCONFLICTING include only mutually compatible operations.
- In INNERCONFLICTING, APPLYCOMPATIBLEGROUP prepares a feature model that includes the effects of both checked operations, CO_a and CO_b, and all their causally preceding operations. Using the same argument as above, we can see that operations in both CP sets are mutually compatible. This also applies to $CP_G(C\,O_a) \cup CP_G(C\,O_b)$ because of the conflict propagation property of $\otimes_O$ (cf. Definition 12). Further, the *no-overwrites* rules have already been checked by SYNTACTICALLYCONFLICTING at this point, therefore $C\,O_a$ and $C\,O_b$ commute. Because $C\,O_a$ and $C\,O_b$ are mutually compatible with their causally preceding operations by definition, the set applied by APPLYCOMPATIBLEGROUP altogether (i.e., $CP_G(C\,O_a) \cup CP_G(C\,O_b) \cup \{C\,O_a, C\,O_b\}$) only includes mutually compatible operations.
- To compute the *uiState* in PROCESSOPERATION and RESOLVECONFLICT (i.e., to derive a feature model to pass to the user interface), APPLYCOMPATIBLEGROUP is invoked with a maximum compatible group from an MCGS, which only contains mutually compatible operations by definition.

Next, we show that our conflict relations $\otimes_I$ and $\otimes_O$ are proper conflict relations (cf. Definition 8), that is, they both must be irreflexive and symmetric (Xue et al. 2003). Further, as we compute both conflict relations algorithmically, the algorithms must be *stable across sites*, meaning that all sites must always compute the same result for two given operations.

We show that $\otimes_I$, when computed by INNERCONFLICTING, is suitable for use within $\otimes_O$ to detect conflicting operations. Because $\otimes_I$ is only used from within $\otimes_O$, we may assume that all causally preceding operations are compatible as guaranteed by conflict propagation. Further, we show that $\otimes_O$, as computed by OUTERCONFLICTING, is also a conflict relation and therefore suitable to be used within the MOVIC algorithm instead of the GRACE conflict relation $\otimes$.

Theorem 3 *Recall that $CO_a \otimes_I CO_b$ if and only if* INNERCONFLICTING*(G, FM_{init}, CO_a, CO_b) = true, where G and FM_{init} are the current CDAG and initial feature model at the site that executes* INNERCONFLICTING. *Further, $CO_a \otimes_O CO_b$ if and only if* OUTERCONFLICTING*(G, CO_a, CO_b) = true, where G is the current CDAG at the site that executes* OUTERCONFLICTING. *Then, $\otimes_I$ and $\otimes_O$ are conflict relations (i.e., symmetric, irreflexive, and stable across sites).*

Proof $\otimes_I$ is stable across sites, that is, INNERCONFLICTING (and SYNTACTICALLYCONFLICTING) always return the same results for two given operations CO_a and CO_b, although the passed CDAG may differ across sites. This is because the CDAG is only used to determine CP sets, which are fully known for CO_a and CO_b (cf. Theorem 2). Further, we have shown above that the preconditions of APPLYCOMPATIBLEGROUP are satisfied in this context, so that its invocation is stable across sites. Altogether, $\otimes_I$ is then stable across sites as well. Furthermore, $\otimes_I$ is irreflexive because INNERCONFLICTING returns *false* if $CO_a = CO_b$. It is also symmetric because SYNTACTICALLYCONFLICTING is used to check both potential execution orders for conflict. Thus, $\otimes_I$ is a conflict relation.

$\otimes_O$ is stable across sites, because the CDAG is only used to determine CIP sets. Because these are subsets of CP sets, they are also fully known (cf. Theorem 2). Further, $\otimes_O$ is irreflexive because OUTERCONFLICTING returns *false* if $CO_a = CO_b$. The symmetry of $\otimes_O$ follows from the symmetry of $\otimes_I$. Thus, $\otimes_O$ is a conflict relation as well. □

From these proofs and previous theorems establishing the correctness of the MOVIC algorithm, we can derive the convergence of our collaborative feature modeling editor.

Theorem 4 *Let $GO = \{O_1, O_2, \ldots, O_n\}$ be a group of operations. Let $MCGS_i$ denote a maximum compatible group set constructed by MOVIC that includes i operations from GO. Then, $MCGS_n$ is the same no matter in which order the n operations are processed. Further, the combined effect (i.e., feature model) derived from $MCGS_n$ is identical regardless of the processing order, that is, the system converges.*

Proof As per our globally targeted object (cf. Section 3.2) and site freeze (cf. Section 3.3) strategies, all operations in our system target always a single feature-model version. Further, $\otimes_O$ is a conflict relation (cf. Theorem 3) and can therefore replace the GRACE conflict relation $\otimes$ in the MOVIC algorithm. Thus, MOVIC constructs the same $MCGS_n$ for GO regardless of the processing order (Sun and Chen 2002, Property 3). The combined effect is then computed by invoking APPLYCOMPATIBLEGROUP on some resulting maximum compatible group, which we have shown to be stable across sites. □

4.3 Intention Preservation

To preserve intentions, a collaborative, real-time editor must ensure that an operation performs its intended execution effect at every site and any time, regardless of its execution context. In other words, its generation and execution contexts always match, where the generation context of an operation consists of the initial feature model and all operations that causally precede the operation; and the execution context may then additionally include operations that do not conflict with any operations in the generation context (Sun et al. 1998). By design, $\otimes_O$ ensures that intentions are preserved by only invoking $\otimes_I$ when it can guarantee that the execution context is appropriate.

However, we have yet to show that the combined effect derived from the MCGS constructed by MOVIC also satisfies this property. The combined effect (i.e., the feature model displayed in the user interface) is computed within PROCESSOPERATION by applying a maximum compatible group CG on the initial feature model using APPLYCOMPATIBLEGROUP. In order to show that for an operation in CG, its execution context matches its generation context, we prove that each operation's CP set is also fully contained in CG. In other words, CG contains no gaps with regard to an operation's causal history.

Theorem 5 *Let G be the CDAG for a group of operations $GO = \{O_1, O_2, \ldots, O_n\}$. Let $MCGS_i$ be any maximum compatible group set constructed by MOVIC that includes i operations from GO. Then, for any $CG \in MCGS_i$ and $O_a \in CG$, $CP_G(O_a) \subseteq CG$.*

Proof First, because O_a has been processed by MOVIC, $CP_G(O_a)$ is fully known (Theorem 2). Now suppose there is any $O_x \in CP_G(O_a)$ that is not in CG. Then there must be an operation $O_b \in CG$ such that $O_x \otimes_O O_b$, otherwise CG would not be a maximum compatible group. Due to the conflict propagation property of $\otimes_O$, also $O_a \otimes_O O_b$. However, $O_a, O_b \in CG$ and $O_a \otimes_O O_b$, therefore CG is not a compatible group, which contradicts the definition of $MCGS_i$. Thus, there is no $O_x \in CP_G(O_a)$ that is not in CG, that is, $CP_G(O_a) \subseteq CG$. □

Furthermore, we desire that no operation is ever rejected, masked, or overridden to improve the collaborators' confidence in the system. We ensure this with the *no-overwrites*, *no-graveyarded*, and *no-group-optional* rules, as described in Section 3.2.

For conflict resolution, we introduced the neutral compatible group to allow cancellation of all pending conflicts. We show that this neutral compatible group is computed the same at all sites and does not contain any gaps, that is, every contained operation's execution context matches its generation context.

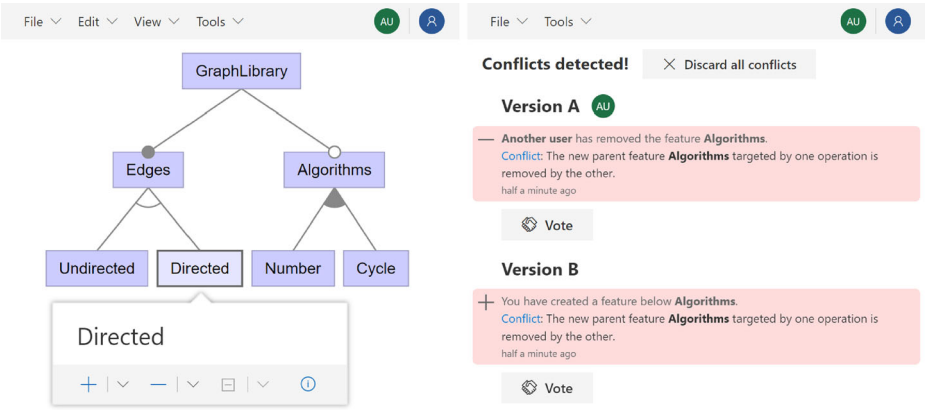
Theorem 6 *Let G and $MCGS$ be the CDAG and maximum compatible group set for a group of operations GO. Further, let $NCG_{MCGS} \coloneqq \bigcap_{CG \in MCGS} CG$ be its neutral compatible group. Then, NCG_{MCGS} is stable across sites. Further, for any $O \in NCG_{MCGS}$, $CP_G(O) \subseteq NCG_{MCGS}$.*

Proof First, NCG_{MCGS} is stable across sites when conflict resolution is initiated. This is because the $MCGS$ is unique (Sun and Chen 2002, Property 1) and converges at all sites (Theorem 4). Now, for any $O \in NCG_{MCGS}$, $O \in CG$ for all $CG \in MCGS$. By Theorem 5, also $CP_G(O) \subseteq CG$ for all $CG \in MCGS$, therefore also $CP_G(O) \subseteq NCG_{MCGS}$. □

From the above theorems, we infer that our collaborative, real-time feature modeling system is CCI-consistent according to Definition 7. As the CCI model has proven itself in practice (Sun and Ellis 1998; Sun et al. 1998), we are confident that our system allows highly-responsive, unconstrained collaboration, while still ensuring basic consistency properties.

5 Implementation

As a proof-of-concept, we have implemented our technique for collaborative, real-time feature modeling in the open-source prototype *variED*. variED is a web-based feature modeling tool that allows multiple users to collaborate on a feature model by visiting the same variED instance in their web browser. In Fig. 5, we depict the user interface of our tool in both phases of the feature-modeling loop as described in Section 3.2.6. We chose a web-based approach because it allows for universal and portable usage of our editor across all platforms, without requiring any setup. In addition, the high degree of abstraction allowed us to implement our prototype in a relatively short time.

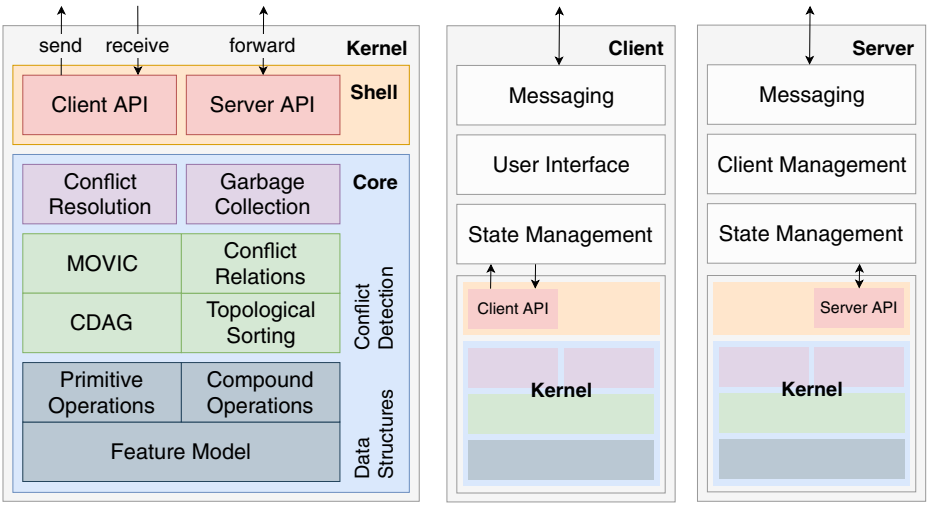


(a) Feature Modeling Phase (b) Conflict Resolution Phase

Fig. 5 Collaborative, real-time feature modeling with variED

When using our editor, collaborators assume a client role and connect to a centralized server, which acts as a message broker for the clients. Thus, our prototype employs a client-server architecture, which allows for simple integration with the web platform and trivially guarantees causality preservation (cf. Section 4). However, our concept is not specific to client-server scenarios and may also be employed in peer-to-peer topologies.

In Fig. 6, we depict the architecture of our prototype, which comprises three components, namely the *collaboration kernel*, *client*, and *server*. The collaboration kernel (cf. Figure 6a) comprises all modules required for collaborative, real-time feature modeling as described in this article. We further provide a client and server component (cf. Figure 6b and c), which



(a) Collaboration Kernel (b) Client (c) Server

Fig. 6 Architecture of variED. Client and server share a collaboration kernel

are executed on the client and server sites, respectively. In the following, we discuss each component in more detail.

The collaboration kernel is primarily concerned with processing operations, that is, detecting and resolving feature modeling conflicts. We chose to implement this component in the *Clojure* programming language (Hickey 2008), which is a Lisp dialect with immutable persistent data structures. Using Clojure integrates well with our client-server infrastructure and allowed us to implement most of our technique rapidly in mathematical pseudocode. To optimize performance, we implemented a garbage collection scheme for pruning the CDAG and introduced a conflict cache that caches the computation results of $\otimes_O$ and $\otimes_I$ to speed up the recursion in OUTERCONFLICTING, the details of which we described previously (Kuiter 2019a). For tracking the causal ordering of operations (cf. Definition 6), we employ the vector clock algorithm developed independently by Fidge (1988) and Mattern (1988).

The mathematical core of the collaboration kernel is surrounded by an imperative shell, which implements Algorithms 6–9 and exposes the *send*, *receive*, and *forward* algorithms as an external API to the client and server component. *Send* is called at a client whenever the user issues an editing operation, which invokes PROCESSOPERATION internally. The data returned by *send* is intended to be sent to the server to serve as input to *forward*, which retrieves an operation at the server, calls PROCESSOPERATION, and returns the operation again, intended to be forwarded to all clients but the originally issuing client. Finally, *receive* accepts such a forwarded operation and calls PROCESSOPERATION as well, possibly creating multiple feature-model versions in the process.

The client component runs at each user's local site inside the web browser. It utilizes TypeScript (Bierman et al. 2014) and ClojureScript (McGranaghan 2011) to provide a local feature-modeling environment, which is connected via a WebSocket (an ordered, real-time messaging channel) to a central server. This central server executes the server component with a Java servlet container, such as Apache Tomcat (Hunter and Crawford 2001). Instead of a user interface, the server component includes a client manager, which allows clients to join and leave collaborative editing sessions.

Our prototype can be deployed on various cloud infrastructure providers, including Amazon AWS, Heroku, and Microsoft Azure to allow for practical usage.[3] In addition, we developed integration and unit test suites for the collaboration kernel and client components. We are confident that the test suites we developed, together with our argumentation in Section 4, confirm the correctness of our prototype.

6 Evaluation

In this section, we report the details of our tool evaluation. To this end, we describe our study design before reporting and discussing its results. We designed and conducted our study based on the guidelines of Wohlin et al. (2012). All artifacts and results are part of our meta repository.[2]

[3]Deployment instructions and a live instance are available in our tool repository.

6.1 Study Design

As we have formally justified the correctness of our technique in Section 4 and showed its feasibility by implementing it (cf. Section 5), our evaluation focused on the usability of our tool (von Nostitz-Wallwitz et al. 2018; Lethbridge et al. 2008; Molich 2010; Macefield 2009). In particular, we aimed to compare our tool to existing collaboration practices. Further, we tried to identify opportunities to improve our implementation, identify core limitations, and collect empirical data that can convince practitioners to use our tool. For this purpose, we defined 15 questions for a survey, which we show in Table 1. First, we asked our participants to provide background information about themselves to help us understand their experiences regarding feature modeling. This included questions on the roles they had during feature modeling (Q_1), their experiences with feature modeling in teaching, studying, academia, and industry (Q_2), and estimates of the feature models' sizes (Q_3). Second, we asked questions regarding collaborative feature modeling. In particular, we were interested in each participant's involvements (Q_4), practices, and strategies (Q_5–Q_8), as well as satisfaction, problems, and limitations of the strategies they employ (Q_9–Q_{11}). Next, each participant had the opportunity to test our tool, either during a live session, by inviting others, or by simulating multi-user editing with multiple browser instances. Finally, we asked the participants to assess our tool (Q_{12}) and describe its limitations as well as benefits compared to current strategies (Q_{13}–Q_{15}). After each of the three sections, all participants also had the option to state additional comments.

As participants, we personally invited researchers and practitioners that we knew worked on feature modeling. To ensure that our tool was used in a collaborative fashion and that we could obtain insights on this aspect, while also aiming to increase the number of responses, we decided to employ two distribution strategies: First, we invited pairs of participants to do live sessions during which two of the authors were present. For this purpose, we designed simple tasks that could cause conflicts (i.e., the participants worked on the same part of an example feature model, e.g., renaming the same feature, changing the same constraints, moving features in conflicting ways). Besides filling in the survey questions, the authors could help the participants with any questions and took notes to document the actions of each participant. So, we obtained additional qualitative insights on the usage of our tool and different strategies to use it. In the end, we conducted four live sessions (each about one hour), resulting in eight responses. Second, we sent out the survey to other invitees with whom it would have been problematic or impossible to coordinate a live session. As aforementioned, these participants had multiple ways to try the collaborative behavior of our tool and we provided our tasks as additional material. We received nine additional responses from these invitations, resulting in a total of 17 surveys being filled out (from participants in Austria, Brazil, France, Germany, Spain, Sweden, and the United States). Even though small, we argue that this sample size is reasonable to understand the usability and major problems of our tool, for three reasons: (1) We are not focusing on statistical tests, but qualitative responses of a comparable, limited number of subjects (i.e., feature-modeling experts), which is why having many participants is less important (Wohlin et al. 2012). (2) We focus on identifying serious problems of our current prototype with a usability study, for which (Molich 2010) actually recommend to involve four to eight participants in each cycle (we conducted one cycle). (3) We found that newer responses (i.e., in they survey) did not identify new serious problems of our technique or disagreements in the satisfaction with our tool (cf. Figure 9), meaning that we achieved saturation as a suitable stop criterion (Wohlin et al. 2012).

Table 1 Survey questions to evaluate our tool

ID	Questions & Answers
	General Feature Modeling Experience
Q_1	What have been your involvements in feature modeling? ☐ Developer ☐ Modeler ☐ Researcher ☐ Domain Expert ☐ Student ☐ Lecturer ☐ Other
Q_2	What is your experience in feature modeling in the following roles? Likert scale (0 - no experience, 5 expert) for roles: teaching, studying, academic, industrial
Q_3	How many features do your feature models contain, on average? ○ <50 ○ 50–100 ○ 100–500 ○ 500+
	Collaborative Feature Modeling Practices
Q_4	What is your experience in collaborative feature modeling? ○ Personally involved ○ Observing/studying ○ Second-hand ○ None ○ Other
Q_5	For what use cases do you use collaborative feature modeling and why? Free text
Q_6	How often do you edit feature models collaboratively? Likert scale (0 - never, 5 - frequently)
Q_7	With how many people do you edit a feature model in a collaborative fashion, on average? Free text
Q_8	What strategy do you employ for collaborative feature modeling and what systems do you use? Free text
Q_9	How satisfied are you with the implemented strategy? ○ Very <un− >satisfied ○ <Un− >Satisfied ○ Slightly <un− >satisfied ○ Not applicable
Q_{10}	What problems do you face during collaborative feature modeling? Free text
Q_{11}	In what use cases do you not apply collaborative feature modeling and why? Free text
	Tool
Q_{12}	How satisfied are you with the tool? ○ Very <un− >satisfied ○ <Un− >Satisfied ○ Slightly <un− >satisfied
Q_{13}	What functionalities of the tool could be improved or are missing with regard to collaborative feature modeling? Free text
Q_{14}	In what use cases would the tool be more suited than your current strategy? Free text
Q_{15}	In what use cases would the tool be less suited than your current strategy? Free text

6.2 Results and Discussion

In the following, we report and discuss the results of our evaluation. Namely, we analyze our participants' experiences with feature modeling, current strategies of collaborative feature modeling, and the feedback on our tool.

6.2.1 Participants

Our 17 participants have been involved in several roles in *general* feature modeling (multiple selections were allowed), mainly as researchers (16), developers (11), and modelers

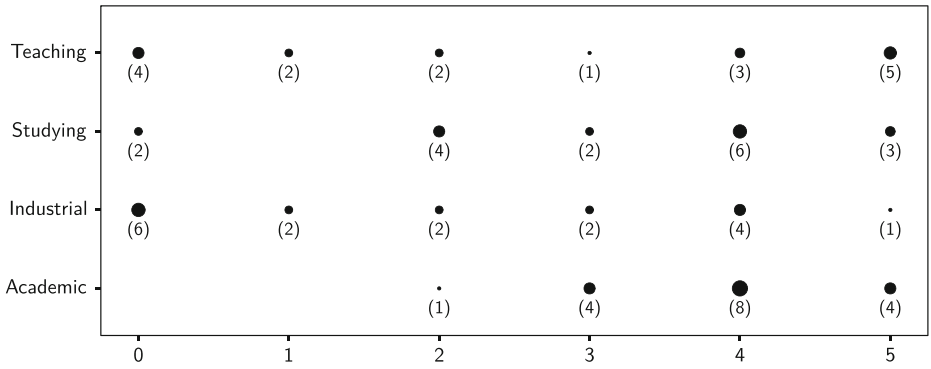


Fig. 7 Experiences of our participants with feature modeling in different roles. Larger circles indicate more responses. The scale ranges from *no experience (0)* to *expert (5)*

(10). In Fig. 7, we show the experiences that our participants have with feature modeling in different roles. We can see that all of them know feature modeling from the academic point of view. Nonetheless, they also have experiences in other roles, most importantly in industrial contexts. On average, our participants worked on smaller feature models with fewer than 50 features (9). However, several have worked on models with 50–100 (5), 100–500 (2), or more than 500 (1) features.

In Fig. 8, we further show to what extent our participants worked with *collaborative* feature modeling. As we can see, six of them did not work or experience collaboration during feature modeling at all. In contrast, six of our participants personally collaborated during feature modeling and four others observed such collaboration. One participant stated to have second-hand experiences, for example, due to a colleague sharing their experiences.

Discussion Overall, we argue that our participants represent a reasonable sample for evaluating our tool. All of them have experiences with feature modeling in general in different roles and contexts, including industry, and have worked with small to medium-sized feature models. Unfortunately, not all of our participants have experiences with collaborative feature modeling. This may limit the insights that we can gain on currently employed strategies for collaboration, but does not affect the feedback we can receive on our tool's usability. As aforementioned, we aimed to obtain feedback on our tool that could convince practitioners

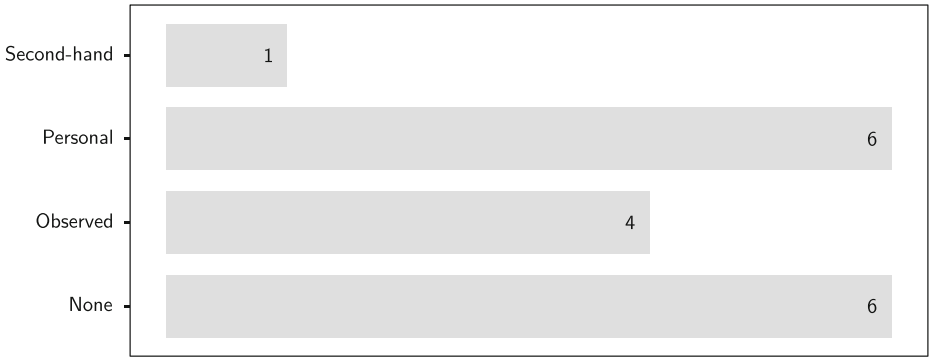


Fig. 8 Our participants' experiences with collaborative feature modeling

to use it. For this purpose, personal experiences and opinions are suitable to understand the usability of our tool and provide convincing arguments.

6.2.2 Collaborative Feature Modeling

Before inquiring concrete strategies for collaborative feature modeling, we asked our participants to exemplify potential use cases for such collaborations and how they work collaboratively on feature models. Our participants stated, for instance, that collaborative feature modeling is helpful for editing independent parts of a feature model (e.g., assigned to different groups of modelers), brainstorming, step-wise refining a model, analyzing requirements, on-the-fly changes, customer support, configuring, teaching, and workshops with domain experts. Essentially, most use cases involve several stakeholders that collaboratively edit the model (e.g., step-wise refinement) or are on different levels of knowledge (e.g., teaching). Moreover, our participants stated that collaborative editing may not occur that frequently (i.e., two responses on level 4 for Q_6, two responses for level 3, and four for level 2), and involves up to 10 stakeholders.

Regarding strategies employed for collaboration prior to using our tool (Q_8), we identified the following:

- *Face-to-Face*: In this strategy, all collaborators must be at the same location. Besides simple discussions or workshops with whiteboards, we found one particularly interesting instance of this strategy: *pair-modeling*. In one of the live-sessions with a team from industry, both participants explained that they would usually edit the feature model while sitting next to each other, and discuss what they would do, analogous to pair-programming (Williams and Kessler 2002). While the face-to-face strategy facilitates communication and collaboration, it has also disadvantages, namely that the number of participants is limited, data must be transferred between workplaces, and meetings must be coordinated. This strategy is known as synchronous, co-located collaboration in the classification of groupware (Baecker et al. 1995).
- *Version-Control Systems*: In this strategy, a version-control system (e.g., Git) is used to collaborate and propagate edits of different stakeholders to the same model. This strategy solves the problems of the face-to-face strategy, but comes with its own problems. For instance, merge conflicts must be manually resolved and can cause severe trouble, which is why parallel editing of the same model parts does not work properly. In the classification of groupware, this strategy is considered a kind of asynchronous, remote collaboration (Baecker et al. 1995).

Overall, these two strategies and their problems resemble those we expected to occur in practice (cf. Section 1) and aimed to solve with our technique.

Discussion Overall, the results on current collaboration practices and strategies align with our initial expectations and existing literature. For instance, our participants confirm that only a small number of stakeholders (ideally) edits a feature model (Fogdal et al. 2016; Berger et al. 2014; Nešić et al. 2019). Moreover, we did identify the strategies for collaborative feature modeling we anticipated, particularly lacking the properties of synchronous, remote collaboration we implemented in our tool (cf. Section 2). So, our tool provides a reasonable addition to current practices by enabling a collaboration strategy that was not available, yet.

6.2.3 Feedback on our Tool

As we had established that our tool addresses an open gap, we were then concerned with its usability and whether our participants considered it to be helpful. In Fig. 9, we compare the satisfaction levels our participants stated for using our tool compared to their current strategy. We remark that the question about satisfaction of the currently employed strategy received fewer responses, as not all participants had experience with collaboration before. Still, five participants were unsatisfied with their current strategy, while our tool received only positive feedback for this evaluation.

Despite the positive feedback, we are aware that our tool is limited in several ways. To assess these limitations, we asked our participants to elaborate on functionalities they missed and on what scenarios our tool would perform better or worse compared to their current strategies, in their opinion. Regarding technical aspects, our participants asked for several convenience functionalities that we were aware of, but did not implement, yet—for instance, highlighting the edits of others, drag and drop, a chat, undo/redo, and different role-defined perspectives. Implementing these convenience functionalities can greatly improve the usability and acceptance of variED. We aim to add these into our prototype, or by integrating our technique into existing tools, such as *FeatureIDE* or *pure::variants*. However, these functionalities are not concerned with our technique itself.

Considering suitable application scenarios for our tool or other strategies, our participants argued in favor of our tool, for instance, for sketching a feature model, discussing changes, remote work, distributed collaboration, structural refactoring, and training. In contrast, our tool faces similar restrictions compared to *Google Docs* or *Overleaf* for text editing, making it less suited for offline work, versioning, and edits on the same parts of the model. Furthermore, variED currently allows any collaborator to change anything, which may not be wanted and can be solved with a role model. One feature that is out of scope for our tool for now, but has been requested several times, is to also support collaborative configuring. We need to address these limitations to make our tool actually usable for practice. Still, the limitations are not concerned with our actual technique, which we can adapt to accordingly in the future, for instance, by submitting offline committed operations when a network is accessible again.

Discussion Overall, our tool received overwhelmingly positive feedback, outperforming current strategies for collaboration. Thus, we argue that our technique provides not only a technical solution for an open problem, but also a valuable tool that can support practitioners and researchers in their work. As we expected, the pros and cons of using our tool align with those of other synchronous, real-time editors that rely on similar concepts. Interestingly, we obtained feedback ranging from our tool can improve “almost all” to “unfortunately few” use cases. In particular, one participant stated that our tool cannot replace face-to-face

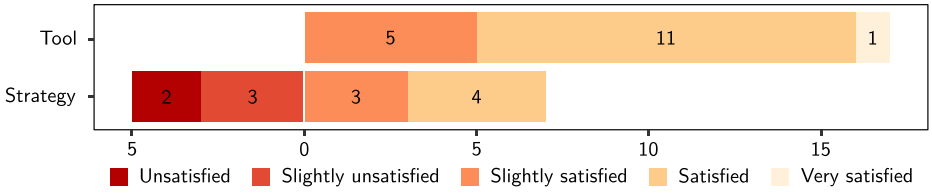


Fig. 9 Participants’ feedback on the satisfaction level of using our tool or the currently employed strategy. There are fewer responses on the current strategy, as not all participants had experiences on collaborative feature modeling beforehand

meetings. However, this was never our goal and we argue that additional communication is needed to properly use our tool. It seems more important to consider how collaboration on feature models occurs in practice and scope the usage of our tool accordingly. Our prototype is missing important convenience functionalities and support for different use cases, limiting its practical usability at the moment. Most of these are extensions that can be build upon our technique and current prototype, which we aim to do in future work.

6.3 Threats to Validity

In the following, we explain the internal and external threats to validity of our evaluation according to Wohlin et al. (2012) and Siegmund et al. (2015).

6.3.1 Internal Validity

One threat to the internal validity of our evaluation is the selection of participants. We personally invited researchers and practitioners who we knew are working on feature modeling, and potentially collaborative feature modeling. So, we limited the number of subjects, and more participants would potentially provide a more detailed and diverse perspective on current collaboration practices for feature modeling and on our tool. However, we aimed to involve interested volunteers to increase motivation and obtain reliable, qualitative data to show the usability of our tool—while also ensuring that our participants were actually experienced in feature modeling. Moreover, we described why we consider our sample reasonable, focusing particularly on achieving saturation to mitigate this threat.

Another threat to the internal validity may be our study design and the questions we asked. Potentially, some questions were misleading or some concepts not clearly explained. To mitigate this threat, we performed a test run with a colleague to evaluate whether all questions were clearly understandable. Furthermore, we first conducted two live sessions (four participants) before deploying our survey, to see whether any formulations or technical limitations caused problems. We slightly extended our explanations accordingly to provide a more helpful documentation, but the questions and tasks seemed to cause no problems. Furthermore, each participant could provide additional comments, also to indicate any problems in understanding the survey or using the tool. However, we received no such comments, indicating that this threat did not cause problems.

Finally, we did not conduct a controlled experiment. So, we did not focus on the internal validity, which we did intentionally to investigate the practical usability of our tool (Siegmund et al. 2015). As a result, we did not evaluate to what extent our tool may perform better (e.g., in terms of task solution time, correctness) compared to other strategies. In contrast, we aimed to obtain qualitative insights and opinions that can convince practitioners to use our tool (von Nostitz-Wallwitz et al. 2018). This may introduce the threat of overly interpreting results that we cannot support with quantitative data. We mitigated this threat by carefully analyzing our qualitative data and avoiding any assessment of its performance compared to other strategies for collaborative feature modeling.

6.3.2 External Validity

Considering the external validity, one threat is that we conducted two different types of evaluations: (1) four live sessions with eight participants during which two authors were present and (2) a survey for which we received nine responses. While both evaluations comprised the same material, descriptions, and questions, only in the live sessions our participants could ask questions to clarify issues. So, the results of both evaluations may not

align perfectly, but we aimed to mitigate this threat by letting all participants answer the survey questions. Moreover, due to the sample size, we cannot generalize the results for all users of collaborative feature modeling; despite ensuring that our participants are knowledgeable in that domain. Due to these two issues, our external validity may be threatened.

Another threat to the external validity that we cannot fully control is the background of our participants. Depending on various factors, such as their work position, feature-modeling experiences, industry collaborations, and motivation, their responses may vary. We aimed to mitigate this threat by inviting participants that have the required knowledge and have different practical experiences with feature modeling (e.g., from industry collaborations). So, we planned to get insights from such different perspectives that allow us to carefully generalize over different backgrounds.

7 Related Work

Closely related to our work is the *CoFM* environment that has been proposed by Yi et al. (2010, 2012). 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, as we only consider a single feature model, which is synchronized among all collaborators. Furthermore, we describe how to detect and resolve conflicting operations, which is not considered in 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 single-user usage (Meinicke et al. 2017; Beuche 2008; Krueger 2007; Mendonça et al. 2009; Acher et al. 2013). To the best of our knowledge, none of the existing tools or techniques supports real-time collaboration. Rather, they allow asynchronous collaboration with version or variation control systems, which is an alternative solution with its own pros and cons compared to our tool.

Linsbauer et al. (2017) classify variation control systems, highlighting a general lack of collaboration support compared to regular version-control systems. In particular, Schwager and Westfechtel (2016, 2017) 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. (2010) introduce *EvoFM*, a technique for modeling variability over time. Their catalog of evolution operators resembles the COs we presented in Section 3.1, but they do not explicitly address collaboration. Similarly, Nieke et al. (2016, 2018) 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. In general, change impact analyses on feature models have been proposed to measure and evaluate conflict potential for modeling decisions (Cho et al. 2011; Hajri et al. 2018; Mazoun et al. 2016; Paskevicius et al. 2012). These techniques do not explicitly address collaboration, but may guide collaborators in their understanding and resolution of conflicts.

8 Conclusion

In this article, we presented a technique for collaborative, real-time feature modeling and conducted an empirical user study to evaluate the usability and usefulness of our corre-

sponding tool, variED. First, we defined the requirements for collaborative, real-time feature modeling, based on the use cases we aimed to support with our tool. To satisfy these requirements, we utilized operation-based editing with a corresponding set of operations and extended the concurrency-control technique MVMD of the graphics editing system GRACE. We adapted the MVMD technique by introducing a conflict relation, a conflict-detection algorithm, and a conflict-resolution strategy that are suitable for feature modeling. In addition, we reasoned about the correctness of our technique according to the CCIs model and showed its feasibility by implementing a prototype. The results of our empirical user study show that our tool supports the defined use cases well and is a helpful means to extend current collaboration strategies. More precisely, the results show that our tool facilitates important use cases that are not covered by currently employed strategies and it received far more positive feedback compared to these strategies, despite its technical limitations.

In future work, we want to address the question of how to raise awareness of collaborators for potentially conflicting editing operations in order to avoid conflicts in the first place. For instance, this could be achieved by facilitating communication between collaborators when they are about to edit the same feature subtree. Besides user studies to evaluate our technique, we also aim to extend our technique with un- and redo operations and more complex COs in order to improve the user experience, incorporating the feedback we received to improve our tool.

A Appendix

In the following, we provide additional details for our operational model, including complete formal descriptions of our primitive and compound operations introduced in Section 3.1, and a proof for Theorem 1 in Section 3.1.3.

A.1 Primitive Operations

PO 2: **updateFeaturePO**(F^{ID}, *attribute*, *oldValue*, *newValue*)

Updates a single attribute of a single feature to a new value. The feature is identified with an existing ID. The old attribute value is also included when the operation is generated to facilitate conflict detection. *Dom* refers to the attribute's domain as defined in Definition 1: $Dom(parentID) = \boldsymbol{ID} \cup \{\bot, \dagger\}$.
Further, $FM.\boldsymbol{F}(F^{ID}).[attribute]$ refers to a particular feature attribute value as specified by *attribute*.

PRECONDITIONS

$F^{ID} \in \boldsymbol{F}^{ID}_{FM}$
$attribute \in \{parentID, optional, groupType, abstract, name\}$
$oldValue = FM.\boldsymbol{F}(F^{ID}).[attribute]$
$newValue \in Dom(attribute)$

POSTCONDITIONS

$FM'.\boldsymbol{F}(F^{ID}).[attribute] = newValue$

PO 3: **createConstraintPO**(C^{ID})

Creates a cross-tree constraint with a globally new unique identifier. Assigns default values to the cross-tree constraint's attributes. Similar to *createFeaturePO*, new cross-tree constraints are initially graveyarded. $\top$ refers to any tautological propositional formula.

PRECONDITIONS	POSTCONDITIONS
$C^{ID} \in \boldsymbol{ID}$	$C^{ID} \in \boldsymbol{C}^{ID}_{FM'}$
$C^{ID} \notin \boldsymbol{C}^{ID}_{FM}$	$FM'.\boldsymbol{C}(C^{ID}).parentID = \dagger$
	$FM'.\boldsymbol{C}(C^{ID}).\phi = \top$

PO 4: **updateConstraintPO**(C^{ID}, *attribute*, *oldValue*, *newValue*)

Updates a single attribute of a single cross-tree constraint to a new value. The cross-tree constraint is addressed with an existing ID. Similar to *updateFeaturePO*, the old attribute value is also included.

PRECONDITIONS	POSTCONDITIONS
$C^{ID} \in \boldsymbol{C}^{ID}_{FM}$	$FM'.\boldsymbol{C}(C^{ID}).[attribute] = newValue$
$attribute \in \{parentID, \phi\}$	
$oldValue = FM.\boldsymbol{C}(C^{ID}).[attribute]$	
$newValue \in Dom(attribute)$	

PO 5: **assertNoChildAddedPO**(F^{ID})

Provides a hint to the conflict detection that concurrent operations must not add children to a feature (cf. Section 3.2). This is required to preserve a single root feature (cf. Definition 3) when removing features. Has no effect when applied on a feature model.

PRECONDITIONS	POSTCONDITIONS
$F^{ID} \in \boldsymbol{F}^{ID}_{FM}$	—

A.2 Compound Operations

CO 2: **createFeatureAbove**(FM, F^{ID}, FC^{IDs})

Creates a feature F^{ID} above a set of sibling features FC^{IDs}. Commonly, the feature is created above only one feature ($\left|FC^{IDs}\right| = 1$).

Require: $F^{ID} \in \boldsymbol{ID}$, $F^{ID} \notin \boldsymbol{F}^{ID}_{FM}$, $FC^{IDs} \neq \emptyset$, $\forall FC^{ID} \in FC^{IDs} : FC^{ID} \in \boldsymbol{F}^{ID}_{FM}$, $\forall FC^{ID}_a, FC^{ID}_b \in FC^{IDs} : FM.\boldsymbol{F}(FC^{ID}_a).parentID = FM.\boldsymbol{F}(FC^{ID}_b).parentID$

▷ the features in FC^{IDs} must be siblings

1: **function** CREATEFEATUREABOVE(FM, F^{ID}, FC^{IDs})
2: $CO \leftarrow [\,]$
3: $FP^{ID} \leftarrow FM.\boldsymbol{F}(FC^{ID}).parentID$ for any $FC^{ID} \in FC^{IDs}$
4: $CO.append(createFeaturePO(F^{ID}))$ ▷ create the new feature
5: $CO.append(updateFeaturePO(F^{ID}, parentID, \dagger, FP^{ID}))$ ▷ insert into tree
6: **if** $FP^{ID} \neq \perp$ **then**
▷ if not creating above the root feature, adjust the group type
7: $CO.append(updateFeaturePO(F^{ID}, groupType, and, FM.\boldsymbol{F}(FP^{ID}).groupType))$
8: **end if**
9: **for all** $FC^{ID} \in FC^{IDs}$ **do**
▷ move the sibling features below the new feature
10: $CO.append(updateFeaturePO(FC^{ID}, parentID, FP^{ID}, F^{ID}))$
11: **end for**
12: **return** CO
13: **end function**

CO 3: **moveFeatureSubtree**(FM, F^{ID}, FP^{ID})

Moves an entire feature subtree rooted at F^{ID} below another feature FP^{ID}. FP^{ID} may neither be part of the moved subtree nor be its root F^{ID}, as this would introduce a cycle to the feature model (cf. Definition 3).

Require: $F^{ID}, FP^{ID} \in \boldsymbol{F}^{ID}_{FM}$, $FP^{ID} \not\preceq_{FM} F^{ID}$

1: **function** MOVEFEATURESUBTREE(FM, F^{ID}, FP^{ID})
2: **return** $[updateFeaturePO(F^{ID}, parentID, FM.\boldsymbol{F}(F^{ID}).parentID, FP^{ID})]$
3: **end function**

CO 4: **removeFeatureSubtree**(FM, F^{ID})

Removes an entire feature subtree rooted at F^{ID} by graveyarding it. Removing the entire feature tree (i.e., F^{ID} is the root feature) is not allowed.

Require: $F^{ID} \in \boldsymbol{F}^{ID}_{FM}$, $FM.\boldsymbol{F}(F^{ID}).parentID \neq \perp$

1: **function** REMOVEFEATURESUBTREE(FM, F^{ID})
2: **return** $[updateFeaturePO(F^{ID}, parentID, FM.\boldsymbol{F}(F^{ID}).parentID, \dagger)]$
3: **end function**

CO 5: **removeFeature**(FM, F^{ID})

Removes a single feature F^{ID}. The removed feature is graveyarded. Its child features FC^{IDs}, if any, are pulled up one level. The root feature can only be removed if it has exactly one child feature (cf. Definition 3), which becomes the new root feature.

Require: $F^{ID} \in \boldsymbol{F}^{ID}_{FM}$, $FM.\boldsymbol{F}(F^{ID}).parentID = \bot \Rightarrow \left|FC^{IDs}\right| = 1$

1: **function** REMOVEFEATURE(FM, F^{ID})
2: $CO \leftarrow [\,]$
3: $FP^{ID} \leftarrow FM.\boldsymbol{F}(F^{ID}).parentID$
4: $FC^{IDs} \leftarrow \{FC^{ID} \in \boldsymbol{F}^{ID}_{FM} \mid FM.\boldsymbol{F}(FC^{ID}).parentID = F^{ID}\}$
5: $CO.append(assertNoChildAddedPO(F^{ID}))$ ▷ later needed for conflict detection
6: **for all** $FC^{ID} \in FC^{IDs}$ **do** ▷ pull child features up
7: $CO.append(updateFeaturePO(FC^{ID}, parentID, F^{ID}, FP^{ID}))$
8: **end for**
9: $CO.append(updateFeaturePO(F^{ID}, parentID, FP^{ID}, \dagger))$ ▷ remove feature
10: **return** CO
11: **end function**

CO 6: **setFeatureOptional**(FM, F^{ID}, $newValue$)

Sets the *optional* attribute of a feature F^{ID} to $newValue$.

Require: $F^{ID} \in \boldsymbol{F}^{ID}_{FM}$, $newValue \in Dom(optional)$

1: **function** SETFEATUREOPTIONAL(FM, F^{ID}, $newValue$)
2: **return** $[updateFeaturePO(F^{ID}, optional, FM.\boldsymbol{F}(F^{ID}).optional, newValue)]$
3: **end function**

CO 7: **createConstraint**(FM, C^{ID}, ϕ_{init})

Creates a cross-tree constraint C^{ID} and initializes it with a given propositional formula ϕ_{init}.

Require: $C^{ID} \in \boldsymbol{ID}$, $C^{ID} \notin \boldsymbol{C}^{ID}_{FM}$, $\phi_{init} \in Dom(\phi)$, $Var(\phi_{init}) \subseteq \boldsymbol{F}^{ID}_{FM}$

1: **function** CREATECONSTRAINT(FM, C^{ID}, ϕ_{init})
2: **return** $[createConstraintPO(C^{ID})$, ▷ create the new constraint
3: $updateConstraintPO(C^{ID}, \phi, \top, \phi_{init})$, ▷ initialize ϕ
4: $updateConstraintPO(C^{ID}, parentID, \dagger, \bot)]$ ▷ remove from graveyard
5: **end function**

CO 8: **setConstraint**(FM, C^{ID}, ϕ_{new})

Sets the propositional formula of a constraint C^{ID} to ϕ_{new}.

Require: $C^{ID} \in \boldsymbol{C}^{ID}_{FM}$, $\phi_{new} \in Dom(\phi)$, $Var(\phi_{new}) \subseteq \boldsymbol{F}^{ID}_{FM}$

1: **function** SETCONSTRAINT(FM, C^{ID}, ϕ_{new})
2: **return** $[updateConstraintPO(C^{ID}, \phi, FM.\boldsymbol{C}(C^{ID}).\phi, \phi_{new})]$
3: **end function**

CO 9: **removeConstraint**(FM, C^{ID})

Removes a cross-tree constraint C^{ID}. The removed cross-tree constraint is graveyarded.
Require: $C^{ID} \in \boldsymbol{C}^{ID}_{FM}$
1: **function** REMOVECONSTRAINT(FM, C^{ID})
2: **return** [$updateConstraintPO(C^{ID}, parentID, FM.\boldsymbol{C}(C^{ID}).parentID, \dagger)$]
3: **end function**

A.3 Proof of Theorem 1

Theorem 1 *Let $FM \in \boldsymbol{FM}$ be a legal feature model. Further, let CO be a compound operation whose preconditions are satisfied with regard to FM. Then, FM' = $applyCO(FM, CO) \in \boldsymbol{FM}$, that is, FM' is again a legal feature model.*

Proof First, *FM'* still has unique identifiers for features and cross-tree constraints: Only compound operations that include *create* POs can affect $\boldsymbol{F}^{ID}_{FM'}$ and $\boldsymbol{C}^{ID}_{FM'}$. For these COs, preconditions such as $F^{ID} \notin \boldsymbol{F}^{ID}_{FM}$ ensure that the created feature or cross-tree constraint has a new unique ID.

Second, *FM'* still contains valid parents (i.e., all referenced *parentIDs* are valid feature IDs, cf. Definition 3): Because *FM* is legal and therefore has valid parents, the only operations that might change this have to create a feature or update a *parentID*. Creating a feature sets its *parentID* to †, which is valid according to Definition 3. As for updating a *parentID*, it can be shown for each compound operation above that the updated *parentID* is still valid. For example, *createFeatureAbove* sets the newly created feature's *parentID* to FP^{ID}, which is a valid feature ID as per precondition $FP^{ID} \in \boldsymbol{F}^{ID}_{FM}$. (The proof is similar for the other operations.)

Third, *FM'* still has valid constraints (i.e., all feature IDs referenced in cross-tree constraints are valid): Because *FM* already has valid constraints and no features are ever deleted, i.e., $\boldsymbol{F}^{ID}_{FM} \subseteq \boldsymbol{F}^{ID}_{FM'}$, we only have to consider *createConstraint* and *setConstraint*, both of which ensure valid constraints as per precondition $Var(\phi) \subseteq \boldsymbol{F}^{ID}_{FM}$.

Next, the root feature condition can only be violated when removing the root feature. We forbid this with preconditions on *removeFeatureSubtree* and *removeFeature*, except for when the root has exactly one child feature. In that case the single root condition is ensured by making that child feature the new root feature.

Finally, *FM'* is still acyclic (i.e., all feature IDs descend from $\perp$ or † in *FM'*): This is simple to show for all compound operations but *moveFeatureSubtree*, where we make use of the precondition that FP^{ID} (the move target) must not descend from F^{ID} (the move source).

Because *FM* is acyclic, both FP^{ID} and F^{ID} descend from $\perp$ or †. As FP^{ID} does not descend from F^{ID} as per precondition, the latter does not appear in the former's path to the root feature. In that case, moving the subtree rooted at F^{ID} below FP^{ID} does not affect FP^{ID}'s path to the root feature. Thus, looking at *FM'*, F^{ID} now descends from FP^{ID} which in turn still descends from $\perp$ or †. By transitivity, F^{ID} also descends from $\perp$ or †, therefore *FM'* is still acyclic. □

Acknowledgments The work of Elias Kuiter, Sebastian Krieter, and Jacob Krüger has been supported by a pure-systems Go SPLC 2019 Challenge project. Jacob Krüger's work has also been supported by an IFI fellowship of the German Academic Exchange Service (DAAD). This research has further been supported by the German Research Foundation (DFG) project EXPLANT (LE 3382/2-3, SA 465/49-3). We would like to thank all of our interview and survey participants for their valuable feedback, and David Broneske for testing our survey.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

Acher M, Collet P, Lahire P, France RB (2013) FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. Sci Comput Program 78(6):657–681

Aho AV, Garey MR, Ullman JD (1972) The Transitive Reduction of a Directed Graph. SIAM J Comput 1(2):131–137

Alves Pereira J, Constantino K, Figueiredo E (2014) A Systematic Literature Review of Software Product Line Management Tools. In: ICSR. Springer, pp 73–89

Apel S, Batory D, Kästner C, Saake G (2013a) Feature-Oriented Software Product Lines. Springer

Apel S, Rhein A, Wendler P, Grösslinger A, Beyer D (2013b) Strategies for Product-Line Verification: Case Studies and Experiments. In: ICSE. IEEE, pp 482–491

Baecker RM, Grudin J, Buxton WAS, Greenberg S (1995) Human-Computer Interaction: Toward the Year 2000. MorganKaufmann

Batory D (2005) Feature Models, Grammars, and Propositional Formulas. In: SPLC. Springer, pp 7–20

Benavides D, Segura S, Ruiz-Cortés A (2010) Automated Analysis of Feature Models 20 Years Later: A Literature Review. Inf Syst 35(6):615–636

Berger T, Rublack R, Nair D, Atlee JM, Becker M, Czarnecki K, Wąsowski A (2013) A Survey of Variability Modeling in Industrial Practice. In: VAMOS. ACM, pp 7:1–7:8

Berger T, Nair D, Rublack R, Atlee JM, Czarnecki K, Wasowski A (2014) Three Cases Of Feature-Based Variability Modeling In Industry. In: MODELS. Springer, pp 302–319

Berger T, Lettner D, Rubin J, Grünbacher P, Silva A, Becker M, Chechik M, Czarnecki K (2015) What is a Feature?: A Qualitative Study of Features in Industrial Software Product Lines. In: SPLC. ACM, pp 16–25

Berlage T, Genau A (1993) A Framework for Shared Applications with a Replicated Architecture. In: UIST. ACM, pp 249–257

Beuche D (2008) Modeling and Building Software Product Lines with Pure::Variants. In: SPLC. IEEE, pp 358–358

Bierman G, Abadi M, Torgersen M (2014) Understanding TypeScript. In: ECOOP. Springer, pp 257–281

Botterweck G, Pleuss A, Dhungana D, Polzer A, Kowalewski S (2010) EvoFM: Feature-Driven Planning of Product-Line Evolution. In: Proceedings of the International Workshop on Product Line Approaches in Software Engineering. ACM, pp 24–31

Bowen TF, Dworack FS, Chow CH, Griffeth N, Herman GE, Lin Y-J (1989) The Feature Interaction Problem in Telecommunications Systems. In: Proceedings of the International Conference on Software Engineering for Telecommunication Switching Systems. IET, pp 59–62

Calder M, Kolberg M, Magill EH, Reiff-Marganiec S (2003) Feature Interaction: A Critical Review and Considered Forecast. Comput Netw 41(1):115–141

Carstensen PH, Schmidt K (1999) Computer Supported Cooperative Work: New Challenges to Systems Design. In: Itoh K (ed) Handbook of Human Factors, pp 619–636

Chen D (2001a) Consistency Maintenance in Collaborative Graphics Editing Systems. Ph.D. Thesis, Griffith University
Chen D, Sun C (2001b) Optional Instant Locking in Distributed Collaborative Graphics Editing Systems. In: Proceedings of the International Conference on Parallel and Distributed Systems. IEEE, pp 109–116
Chen D, Sun C (2001c) Undoing Any Operation in Collaborative Graphics Editing Systems. In: Proceedings of the International Conference on Supporting Group Work. ACM, pp 197–206
Chen L, Babar MA (2011) A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. Inf Softw Technol 53(4):344–362
Cho H, Gray J, Cai Y, Wong S, Xie T (2011) Model-Driven Impact Analysis of Software Product Lines. In: Osis J, Asnina E (eds) Model-Driven Domain Analysis and Software Development: Architectures and Functions. IGI Global, pp 275–303
Cormack GV (1995) A Counterexample to the Distributed Operational Transform and a Corrected Algorithm for Point-to-Point Communication. Technical Report CS-95-08
Czarnecki K (2013) Variability in Software: State of the Art and Future Directions. In: FASE. Springer, pp 1–5
Czarnecki K, Grnbacher P, Rabiser R, Schmid K, Wąsowski A (2012) Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In: VAMOS. ACM, pp 173–182
Dennis AR, Pootheri SK, Natarajan VL (1998) Lessons from the Early Adopters of Web Groupware. J Manag Inf Syst 14(4):65–86
Dewan P (1999) Architectures for Collaborative Applications. Comput Supported Coop Work 7:169–193
Durán A, Benavides D, Segura S, Trinidad P, Ruiz-Cortés A (2017) FLAME: A Formal Framework for the Automated Analysis of Software Product Lines Validated by Automated Specification Testing. Softw Syste Model 16(4):1049–1082
Ellis C, Gibbs S (1989) Concurrency Control in Groupware Systems. In: Proceedings of the International Conference on Management of Data. ACM, pp 399–407
Ellis C, Gibbs S, Rein G (1991) Groupware: Some Issues and Experiences. CACM 34(1):39–58
Elmasri R, Navathe S (2010) Fundamentals of Database Systems. AddisonWesley
Felfernig A, Benavides D, Galindo JA, Reinfrank F (2013) Towards Anomaly Explanation in Feature Models. In: Proceedings of the International Configuration Workshop, pp 117–124
Fidge CJ (1988) Timestamps in Message-Passing Systems that Preserve the Partial Ordering. Austral Comput Sci Commun 10(1):56–66
Fogdal T, Scherrebeck H, Kuusela J, Becker M, Zhang B (2016) Ten Years of Product Line Engineering at Danfoss: Lessons Learned and Way Ahead. In: SPLC. ACM, pp 252–261
Gibbs S (1989) LIZA: An Extensible Groupware Toolkit. In: Proceedings of the Conference on Human Factors in Computing Systems. ACM, pp 29–35
Greenberg S (1991) Personalizable Groupware: Accommodating Individual Roles and Group Differences. In: ECSCW. Springer, pp 17–31
Greenberg S, Marwood D (1994) Real Time Groupware as a Distributed System: Concurrency Control and itss Effect on the Interface. In: CSCW. ACM, pp 207–217
Grudin J (1994) Computer-Supported Cooperative Work: History and Focus. CACM 27(5):19–26
Hajri I, Goknil A, Briand LC, Stephany T (2018) Change Impact Analysis for Evolving Configuration Decisions in Product Line Use Case Models. J Syst Softw 139:211–237
Hickey R (2008) The Clojure Programming Language. In: DLS. ACM, pp 1
Horcas J-M, Pinto M, Fuentes L (2019) Software Product Line Engineering: A Practical Experience. In: SPLC. ACM, pp 164–176
Hunter J, Crawford W (2001) Java Servlet Programming: Help for Server Side Java Developers. OReilly
Imine A, Rusinowitch M, Oster G, Molli P (2006) Formal Design and Verification of Operational Transformation Algorithms for Copies Convergence. Theor Comput Sci 351(2):167–183
Johansen R (1991) Groupware: Future Directions and Wild Cards. J Organ Comput 1(2):219–227
Kahn AB (1962) Topological Sorting of Large Networks. CACM 5(11):558–562
Kowal M, Ananieva S, Thüm T (2016) Explaining Anomalies in Feature Models. In: GPCE. ACM, pp 132–143
Krüger J, Mahmood W, Berger T (2020) Promote-pl: A Round-Trip Engineering Process Model for Adopting and Evolving Product Lines. In: SPLC. ACM, pp 2:1–2:12
Krueger CW (2007) BigLever Software Gears and the 3-Tiered SPL Methodology. In: OOPSLA. ACM, pp 844–845
Kuiter E (2019a) Consistency Maintenance for Collaborative Real-Time Feature Modeling. Bachelor Thesis, University of Magdeburg
Kuiter E, Krieter S, Krüger J, Leich T, Saake G (2019b) Foundations of Collaborative, Real-Time Feature Modeling. In: SPLC. ACM, pp 257–264

Lamport L (1978) Time, Clocks, and the Ordering of Events in a Distributed System. CACM 21(7):558–565
Lethbridge TC, Lyon S, Perry P (2008) The Management of University–Industry Collaborations Involving Empirical Studies of Software Enginee. In: Shull F, Singer J, Sjøberg DIK (eds) Guide to Advanced Empirical Software Engineering. Springer, pp 257–281
Linsbauer L, Berger T, Grünbacher P (2017) A Classification of Variation Control Systems. In: GPCE. ACM, pp 49–62
Macefield R (2009) How to Specify the Participant Group Size for Usability Studies: A Practitioners Guide. J Usability Stud 5(1):34–45
Manz C, Stupperich M, Reichert M (2013) Towards Integrated Variant Management In Global Software Engineering: An Experience Report. In: ICGSE. IEEE, pp 168–172
Mattern F (1988) Virtual Time and Global States of Distributed Systems. In: Proceedings of the International Workshop on Parallel and Distributed Algorithms. North Holland, pp 215–226
Mazoun J, Bouassida N, Ben-Abdallah H (2016) Change Impact Analysis for Software Product Lines. J King Saud Univ Comput Inf Sci 28(4):364–380
McGranaghan M (2011) ClojureScript: Functional Programming for JavaScript Platforms. IEEE Internet Computing 15(6):97–102
Meinicke J, Thüm T, Schröter R, Benduhn F, Leich T, Saake G (2017) Mastering Software Variability with FeatureIDE. Springer
Mendonça M, Branco M, Cowan D (2009) S.P.L.O.T.: Software Product Lines Online Tools. In: OOPSLA. ACM, pp 761–762
Morris MR, Ryall K, Shen C, Forlines C, Vernier F (2004) Beyond "Social Protocols": Multi-user Coordination Policies for Co-Located Groupware. In: CSCW. ACM, pp 262–265
Molich R (2010) A Critique of "How to Specify the Participant Group Size for Usability Studies: A Practitioner's Guide". J Usability Stud 5(3):124–128
Nešić D, Krüger J, Stanciulescu S, Berger T (2019) Principles of Feature Modeling. In: ESECFSE. ACM, pp 62–73
Nieke M, Seidl C, Schuster S (2016) Guaranteeing Configuration Validity in Evolving Software Product Lines. In: VAMOS. ACM, pp 73–80
Nieke M, Seidl C, Thüm T (2018) Back to the Future: Avoiding Paradoxes in Feature-Model Evolution. In: SPLC. ACM, pp 48–51
Oster G, Molli P, Urso P, Imine A (2006) Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems. In: Proceedings of the International Conference on Collaborative Computing. IEEE, pp 1–10
Paskevicius P, Damasevicius R, Łtuikys V (2012) Change Impact Analysis of Feature Models. In: Proceedings of the International Conference on Information and Software Technologies. Springer, pp 108–122
Pohl K, Böckle G, van der Linden F (2005) Software Product Line Engineering: Foundations, Principles and Techniques. Springer
Prakash A (1999) Group Editors. In: Beaudouin-Lafon M (ed) Computer Supported Co-Operative Work. Wiley, pp 103–134
Randolph A, Boucheneb H, Imine A, Quintero A (2013) On Consistency of Operational Transformation Approach. Electron Proc Theor Comput Sci 107:45–59
Schobbens P-Y, Heymans P, Trigaux J-C (2006) Feature Diagrams: A Survey and a Formal Semantics. In: Proceedings of the International Requirements Engineering Conference. IEEE, pp 139–148
Schobbens P-Y, Heymans P, Trigaux J-C, Bontemps Y (2007) Generic Semantics of Feature Diagrams. Comput Netw 51(2):456–479
Schollmeier R (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, pp 101–102
Schwarz R, Mattern F (1994) Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. Distrib Comput 7(3):149–174
Schwägerl F, Westfechtel B (2016) Collaborative and Distributed Management of Versioned Model-Driven Software Product Lines. In: Proceedings of the International Joint Conference on Software Technologies. SciTePress, pp 83–94
Schwägerl F, Westfechtel B (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, pp 15–28
Shapiro M, Preguia N, Baquero C, Zawirski M (2011) Conflict-Free Replicated Data Types. In: Stabilization, Safety, and Security of Distributed Systems. Springer, pp 386–400

Siegmund J, Siegmund N, Apel S (2015) Views on Internal and External Validity in Empirical Software Engineering. In: ICSE. IEEE, pp 9–19
Stefik M, Foster G, Bobrow DG, Kahn K, Lanning S, Suchman L (1987) Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings. CACM 30(1):32–47
Sun C, Ellis C (1998) Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. In: CSCW. ACM, pp 59–68
Sun C, Jia X, Zhang Y, Yang Y, Chen D (1998) Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems. TOCHI 5(1):63–108
Sun C, Sosič R (1999) Consistency Maintenance in Web-Based Real-Time Group Editors. In: Proceedings of the International Conference on Distributed Computing Systems Workshops on Electronic Commerce and Web-Based Applications. IEEE, pp 15–22
Sun C, Chen D (2000) A Multi-Version Approach to Conflict Resolution in Distributed Groupware Systems. In: Proceedings of the International Conference on Distributed Computing Systems. IEEE, pp 316–325
Sun C, Chen D (2002) Consistency Maintenance in Real-Time Collaborative Graphics Editing Systems. TOCHI 9(1):1–41
Sun D, Xia S, Sun C, Chen D (2004) Operational Transformation for Collaborative Word Processing. In: CSCW. ACM, pp 437–446
Sun C, Xia S, Sun D, Chen D, Shen H, Cai W (2006) Transparent Adaptation of Single-User Applications for Multi-User Real-Time Collaboration. TOCHI 13(4):531–582
Sun C, Yang Y, Zhang Y, Chen D (1996) A Consistency Model and Supporting Schemes for Real-Time Cooperative Editing Systems. In: Proceedings of the Australian Computer Science Conference, pp 582–591
Sun C, Xu Y, Agustina A (2014) Exhaustive Search of Puzzles in Operational Transformation. In: CSCW. ACM, pp 519–529
von Nostitz-Wallwitz I, Krüger J, Siegmund J, Leich T (2018) Knowledge Transfer from Research to Industry: A Survey on Program Comprehension. In: ICSEC. IEEE, pp 300–301
Williams L, Kessler R (2002) Pair Programming Illuminated. Addison Wesley
Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in Software Engineering. Springer
Wulf V (1995) Negotiability: A Metafunction to Tailor Access to Data in Groupware. Behav Inf Technol 14(3):143–151
Xue L, Orgun M, Zhang K (2003) A Multi-Versioning Algorithm for Intention Preservation in Distributed Real-Time Group Editors. In: Proceedings of the Australasian Computer Science Conference. ACS, pp 19–28
Yi L, Zhang W, Zhao H, Jin Z, Mei H (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, pp 23:1–23:4
Yi L, Zhao H, Zhang W, Jin Z (2012) CoFM: An Environment for Collaborative Feature Modeling. In: Proceedings of the International Requirements Engineering Conference. IEEE, pp 317–318

Elias Kuiter is a Master student at the Otto-von-Guericke University Magdeburg. He received his B.Sc. degree in Computer Science at the Otto-von-Guericke University in 2019 and has been working as a research assistant at the Databases and Software Engineering work group. His current research focuses on the application of formal methods to software product lines, and collaborative feature modeling in particular.

Sebastian Krieter is a PhD student at the Otto-von-Guericke University Magdeburg and associated researcher at the Harz University of Applied Sciences. In 2015 he received his M.Sc. degree in Informatics at the Otto-von-Guericke University and has since been working as research associate at the Otto-von-Guericke University Magdeburg, Germany and later at the Harz University of Applied Sciences Wernigerode, Germany. In his research he investigates techniques and analyses for variability modeling, configuration management, and software product line testing.

Jacob Krüger is a PhD student and associated researcher at the Databases and Software Engineering work group of the Otto-von-Guericke University Magdeburg. He received his M.Sc. degree in Business Informatics at the Otto-von-Guericke University in 2016, has been working as research associate at the Harz University of Applied Sciences Wernigerode, and visited Chalmers — University of Gothenburg in Sweden as well as the University of Toronto in Canada. His research addresses feature-oriented software development, with particular focus on software evolution, program comprehension, and human factors.

Gunter Saake received his diploma and PhD in Computer Science from the Technical University of Braunschweig, F.R.G., in 1985 and 1988, respectively. From 1988 to 1989, he was a visiting scientist at the IBM Heidelberg Scientific Center, where he joined the Advanced Information Management project and worked on language features and algorithms for sorting and duplicate elimination in nested relational database structures. In January 1993, he received the Habilitation degree (venia legendi) for Computer Science from the Technical University of Braunschweig. Since May 1994, Gunter Saake is a fulltime professor for Databases and Information Systems at the Otto-von-Guericke University, Magdeburg. His research interests include database integration, tailor-made data management, object-oriented information systems, and information fusion.

Thomas Leich is Professor for Requirements Engineering at Harz University of Applied Sciences in Wernigerode, Germany. He is also executive director of METOP GmbH, an affiliate institute to the University of Magdeburg. Since 2001, he worked for several DAX 30 companies as consultant and software architect. In 2004, he initiated FeatureIDE as a part of the FeatureC++ project at the University of Magdeburg. Until today, he is responsible for industrial extensions and consulting of FeatureIDE.

Affiliations

Elias Kuiter[1] · Sebastian Krieter[1,2] · Jacob Krüger[1,3] · Gunter Saake[1] · Thomas Leich[2]

Elias Kuiter
kuiter@ovgu.de

Jacob Krüger
jkrueger@ovgu.de

Gunter Saake
saake@ovgu.de

Thomas Leich
tleich@hs-harz.de

1 Otto-von-Guericke University Magdeburg, Magdeburg, Germany

2 Harz University of Applied Sciences Wernigerode, Wernigerode, Germany

3 University of Toronto, Toronto, ON, Canada