University of Magdeburg

School of Computer Science

Bachelor Thesis

Consistency Maintenance for Collaborative Real-Time Feature Modeling

Author:

Elias Kuiter

April 1, 2019

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake

M.Sc. Sebastian Krieter

M.Sc. Jacob Krüger

Otto-von-Guericke University Magdeburg

Department of Technical and Business Information Systems

Kuiter, Elias:
Consistency Maintenance for Collaborative Real-Time Feature Modeling
Bachelor Thesis, University of Magdeburg, 2019.

Abstract

Feature modeling is a core activity in software product line engineering to manage and maintain variability in highly configurable software systems. In the feature modeling process, collaboration among stakeholders is vital because domain knowledge is typically spread across different domain experts. However, state-of-the-art feature modeling tools only support single-user editing. While version and variation control systems offer collaboration support to some degree, they promote divergence and merge conflicts. To facilitate simultaneous editing and allow instant feedback from other collaborators, real-time collaboration support is needed.

In this thesis, we lay the formal foundation of collaborative real-time feature modeling. We focus on consistency maintenance, that is, ensuring the syntactic and semantic consistency of edited feature models at all times. Thus, our primary concern is detecting and resolving conflicts between simultaneous actions of different collaborators. To this end, we identify a suitable concurrency control technique and adapt it to the feature modeling domain. We reason about the correctness of our approach and demonstrate its feasibility by implementing it in a web-based prototype. Our design leaves room for future extensions and provides the basis for enhancing single-user feature modeling tools with support for real-time collaboration.

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
CAD	Computer-Aided Design
CCI	Convergence, Causality & Intention Preservation
CDAG	Causal Directed Acyclic Graph
CG	Compatible Group
CGS	Compatible Group Set
CIP	Causally Immediately Preceding
CO	Compound Operation
CP	Causally Preceding
CRDT	Conflict-Free Replicated Data Type
CSCW	Computer-Supported Cooperative Work
FM	Feature Model
GRACE	Graphics Collaborative Editing
MCG	Maximum Compatible Group
MCGS	Maximum Compatible Group Set
MOVIC	Multiple Object Versions Incremental Creation
MVMD	Multi-Version Multi-Display
MVSD	Multi-Version Single-Display
OT	Operational Transformation
P2P	Peer-to-Peer
PO	Primitive Operation
SPL	Software Product Line
UML	Unified Modeling Language
UUID	Universally Unique Identifier
variED	Variability Editor

1. Introduction

In today's software industry, customizable software systems are increasingly demanded [FLLE14, PBvdL05]. *Software product lines* are a systematic approach to develop and manage such highly configurable, software-intensive systems. A software product line allows developers and stakeholders to construct customized software products from reusable artifacts. Customers select their particular desired functionalities and then derive a customized product that includes these functionalities in an automated fashion. Besides mass customization, the adoption of software product lines can reduce development costs and sustain maintainability of highly configurable software systems [CN02, KMB+01, PBvdL05].

Software product lines are usually organized around composable blocks of functionality, so-called *features* [ABKS13, KCH+90]. A feature represents some characteristic of a software system, which a customer may or may not desire in the customized product. Further, features may have complex dependencies among each other (e.g., depending on or excluding other features), which must be respected in any customized product. Managing the set of features and their dependencies is a core activity in developing and maintaining a software product line, referred to as *feature modeling*. In feature modeling, a *feature model* is constructed, which describes the variability represented by a software product line and defines the commonalities and differences between customized products. The feature model is central to a software product line, as it describes all customizable products, implementation details and domain knowledge of stakeholders. Thus, it provides a layer of abstraction comprehensible for customers, developers and stakeholders alike. Accordingly, constructing and maintaining a feature model is a critical task in software product line engineering [ABKS13, BNR+14].

Feature modeling is inherently collaboration-intensive because, typically, many stakeholders contribute their expertise and knowledge to the modeling process. Further, stakeholders (such as domain experts and customers) communicate and coordinate with each other to determine the scope of a software product line and construct the actual feature model [BBS10, KCH+90]. This is necessary as in highly configurable software systems, a single stakeholder's expertise is usually limited to a portion of

the modeled variability. Consequently, the domain knowledge about a software product line is typically spread across different stakeholders [YZZJ12, ZYZJ13]. Thus, stakeholders are faced with two problems [YZZJ12]: First, natural language is too imprecise for feature modeling and promotes confusion among collaborators, as informal specifications of features and feature dependencies may be ambiguous. Second, collaborating stakeholders may accidentally duplicate or conflict with other collaborators' work. In particular, determining and resolving conflicts among stakeholders is an unstructured, cumbersome process. Addressing these problems is necessary to allow stakeholders to collaborate efficiently when constructing a feature model.

Existing tools like *FeatureIDE* [MTS+17], *pure::variants* [Beu08], or *Gears* [Kru07] do not support users in collaborative feature modeling. Instead, the state of the art for collaboration on feature models is to use version or variation control systems [LBG17, O'S09]. However, such systems do not allow for simultaneous editing of feature models; further, they promote divergence and merge conflicts. Thus, explicit support for real-time collaboration is needed. Real-time collaborative editing systems such as *Google Docs* or *Overleaf* can enhance collaborative activities, as they provide a shared workspace that allows for tight collaboration [Pra99]. In addition, they allow for simple sharing of edited documents and visualization of collaborator activity [JLK17]. We see various potential use cases for a collaborative real-time feature modeling system, for example:

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

In such scenarios, a collaborative real-time feature modeling system allows users to leverage group synergies and solve modeling tasks that would be difficult for individuals. Further, such a system ensures effective and efficient editing of feature models in the presence of multiple collaborators. To approach the problem of how to build a collaborative real-time feature modeling system, we identified the following research questions:

RQ_1 How can we manage concurrency from a technical standpoint? That is, what are concepts to ensure a consistent state of the feature model in the presence of multiple, real-time collaborators?

RQ_2 What (inter-)actions can potentially cause conflicts?

RQ_3 How can we resolve conflicting actions while ensuring user satisfaction?

With our work, we intend to give initial answers to these research questions.

Contributions

This thesis aims to provide a first concept as well as a working prototype for a real-time collaborative feature model editor. Our concept adapts and extends existing techniques, to the end of answering our research questions. Further, it provides the basis to enhance existing tools with collaboration support to enable the aforementioned use cases. In particular, we make the following contributions:

- We identify what requirements a collaborative real-time feature model editor should satisfy and choose a suitable concurrency control technique.
- We extend this concurrency control technique to allow for collaborative real-time feature modeling. In particular, we introduce strategies and mechanisms to detect and resolve emerging conflicts.
- We reason about the correctness of our approach and implement it in a web-based prototype to demonstrate its feasibility.

Our concept for collaborative real-time feature modeling is aimed at small groups of collaborators (e.g., 3–4) that edit a single feature model at the same time. We rely on an existing optimistic concurrency control algorithm, which immediately executes locally generated operations to allow for fast and responsive feature modeling. When faced with conflicting operations, this algorithm preserves multiple collaborators' editing intentions in multiple feature model versions. With our extensions, the algorithm can ensure basic syntactic consistency of feature models. Further, almost arbitrary semantic properties on feature models can be preserved (e.g., that the feature model may not contain redundant feature dependencies). In case of conflict, collaborators are prompted to resolve their conflict manually by means of a voting technique. Thus, the introduction of undesirable anomalies into the feature model can be avoided.

In this thesis, we focus on the conceptual foundations of collaborative real-time feature modeling. Our prototype serves as a proof of concept to guide implementation of the concept in existing tools, such as *FeatureIDE* and *pure::variants*, and may be used in future evaluations.

Outline

The thesis is structured as follows: In Chapter 2, we provide relevant background knowledge on software product lines and computer-supported cooperative work. In Chapter 3, we introduce a representation of feature models and modeling operations, both of which are suitable for collaborative feature modeling. In Chapter 4, we analyze requirements for collaborative feature modeling and identify a suitable concurrency control technique. Subsequently, in Chapter 5 we extend this technique to allow for collaborative feature modeling, focusing on conflict detection and resolution. In Chapter 6, we present our prototypical implementation, focusing on the integration of our concurrency control technique. We discuss work related to this thesis in Chapter 7. Finally, we conclude and list opportunities for future work in Chapter 8.

2. Background

Collaborative real-time feature modeling involves two major research areas, *software product lines* and *computer-supported cooperative work*. In this chapter, we introduce key concepts from both fields that are relevant to this thesis. First, we discuss *software product lines*, focusing on feature modeling, feature diagrams, and analyses on feature models. Further, we examine the field of *computer-supported cooperative work*, which lays the groundwork for multi-user collaboration software. In particular, we cover collaborative real-time editors and how to maintain consistency in such editors. Finally, we consider other research areas such as version control systems and how they relate to collaborative real-time editors.

2.1 Software Product Lines

A Software Product Line (SPL) is "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [CN02]. With SPLs, software engineers can systematically construct software systems from reusable artifacts (e.g., code or documentation). Within this process, the customer's specific requirements are taken into consideration to derive tailored solutions, also referred to as *products* of the SPL [ABKS13]. *Software product line engineering* comprises all development and management activities for adopting and applying software product lines.

The appeal of SPLs lies in a number of possible benefits [ABKS13, KMB+01, PBvdL05]: In particular, the development of new software products can leverage similarities with existing products in the SPL, which leads to lower development costs and faster time to market. Another advantage is that SPLs help to manage complexity and sustain maintainability, which improves the overall quality of the products. The major downside to SPLs is the increased initial planning effort, which only pays off after a certain number of products have been developed based on the SPL [ABKS13]. Further, extracting an SPL from a set of legacy software systems carries a financial risk [CK02, KFM+16].

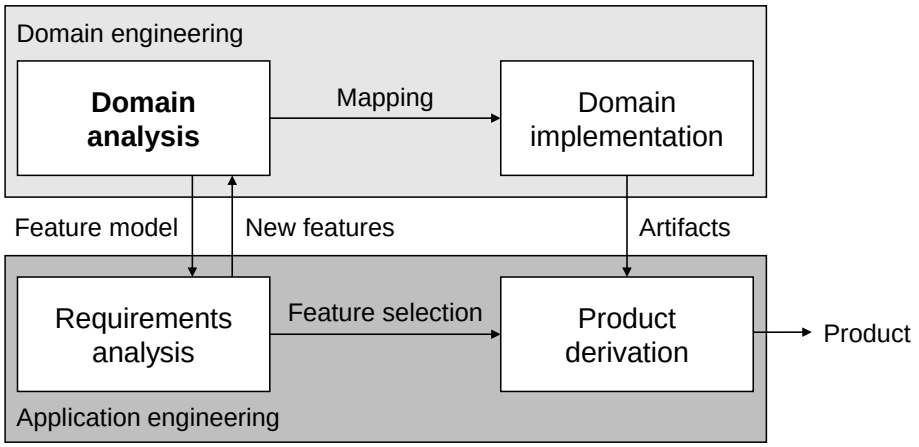


Figure 2.1: Software Product Line (SPL) development process [ABKS13]. In this thesis, we focus on domain analysis, feature modeling in particular.

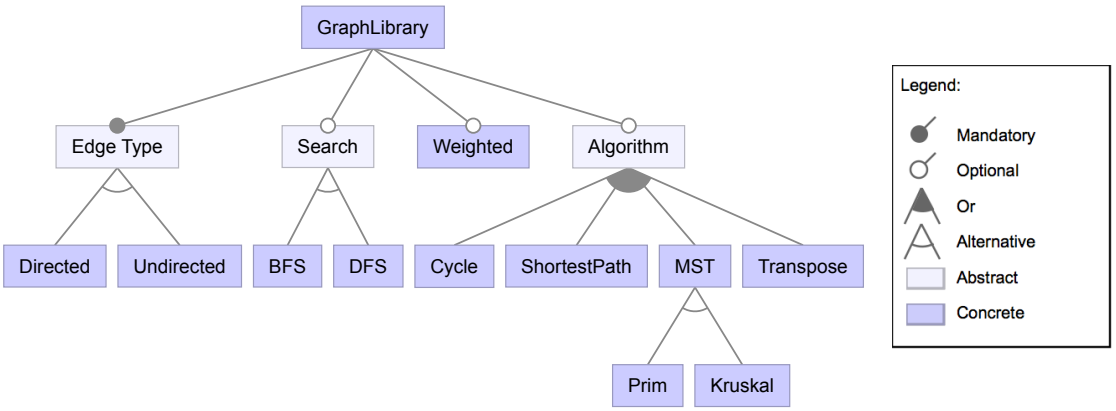
SPLs are commonly organized around the concept of *features*. A feature is a "characteristic or end-user-visible behavior of a software system" [ABKS13, KCH+90]. In a feature-oriented SPL, features are used to define commonalities as well as differences between software products. Features play a major role in every activity of the SPL development process, as they are the building blocks to define, implement, and deliver variability. Our work focuses on these *feature-oriented software product lines*, which are common in research and industry [BRN+13, CB11, SRC+12]. Other variability modeling techniques are compared by Czarnecki et al. [CGR+12].

Czarnecki and Eisenecker have proposed a development process for SPLs, which we display in Figure 2.1 [ABKS13, Bö04, CE00, PBvdL05]. This process distinguishes four basic activities in SPL engineering, which we discuss briefly. In a *domain analysis*, the scope of the SPL is defined. This includes which features and, by extension, which products the SPL comprises. In feature-oriented SPLs, a *feature model* captures the results of this analysis. Then, as part of the *domain implementation*, features are mapped to reusable artifacts (such as source code or documentation). To develop a new software product, the customer performs a *requirements analysis*, in which she selects desired features from the feature model. By then reusing the artifacts developed in the domain implementation, *product derivation* can be performed to obtain the desired product. In this thesis, we focus on collaborative domain analysis, which is concerned with designing a feature model.

2.1.1 Feature Modeling

Feature modeling refers to a critical activity of the SPL development process. In this process, engineers determine and model the variability of an SPL as part of a domain analysis. This is often necessary, as features can have complex relationships. For example, selecting a feature may require or exclude other features. These dependencies have to be satisfied to be able to derive a product that works as intended.

Feature Models (FMs), the artifacts constructed in feature modeling, define the valid products of an SPL by specifying features and their relationships. Multiple representations for feature models have been discussed in the literature, including



$$MST \Rightarrow Undirected \wedge Weighted$$
$$Cycle \Rightarrow Directed$$

Figure 2.2: Feature diagram of the graph product line.

feature diagrams, propositional formulas, and grammars [Bat05]. In this thesis, we focus on feature diagrams, a graphical representation of feature models [CE00, KCH+90, SHT06].

Feature diagrams are a popular notation for feature models [BRN+13]. A feature diagram consists of a feature tree and a set of cross-tree constraints. A *feature tree* is a graphical representation of all features and most of their dependencies. In a feature tree, nodes represent features, and edges model the different kinds of relationships between features. In Figure 2.2, we show an example of a feature diagram that models variability for a software library that operates on graph structures [ABKS13, LHB01]. The *graph product line* includes the features `GraphLibrary`, `Edge Type`, `Directed` etc., represented as nodes in the tree. Special notation is available to express common feature relationships. In particular, there are edges that model parent-child relationships in different ways: Optional features may or may not be selected, provided that their parent is selected; while mandatory features are additionally required if their parent is selected. In our example, the `Search`, `Weighted`, and `Algorithm` features are optional, requiring only the root feature `GraphLibrary` to be selected. The `Edge Type` feature, however, is mandatory, so it is required whenever the root feature is selected. The root feature is special in that it always has to be selected in any product, i.e., it is always mandatory. Additional *feature groups* allow more complex dependencies to be expressed: A feature with an *or* group requires at least one of its child features to be selected, while a feature with an *alternative* group requires that *exactly* one child feature is selected, which models mutual exclusion of features. For example, as edges in a graph can only be `Directed` or `Undirected`, but not both, these features are part of an *alternative* group below the `Edge Type` feature. In contrast, the `Algorithms` feature requires at least one, but allows multiple algorithms to be selected, because it has an *or* group. Commonly, a distinction is made between abstract (e.g., `Edge Type`) and concrete (e.g., `Directed`) features: Abstract features are only used to structure the feature tree, whereas concrete features actually affect the generated

product. This distinction is useful to improve readability, but is not necessary for modeling variability. We include it nonetheless, because it is a common example for an additional attribute on features, which our concept also supports.

Building the tree structure requires a hierarchical decomposition of the SPL's variability [TOHS99]. As features may have arbitrary relationships with each other, the question arises: How to model the remaining dependencies that cannot be accommodated by the chosen tree structure? One solution to this problem is to introduce dedicated *require* and *exclude* edges, but it has been shown that this is not sufficient to express industrial-scale feature models [KTM+17]. Another, more flexible solution is to introduce cross-tree constraints. *Cross-tree constraints* may specify arbitrary constraints over any features in the feature tree. They may be expressed as propositional formulas. In the example, the `MST` feature (which is used to calculate minimum spanning trees) only makes sense in the context of graphs which are `Undirected` and `Weighted`, which is expressed below the feature tree as a cross-tree constraint.

Feature diagrams have the advantage that they are readable and maintainable for humans. This makes them suitable for communicating domain knowledge with stakeholders and for feature modeling tasks, which is why they are widely used in practice [BGR+17, BRN+13]. There are other kinds of feature diagrams that attach metadata to features (*extended feature models*) [BTRC05], allow arbitrary cardinalities for groups [CHE04], or model temporal or contextual variability [NES17, NSS16]. In this thesis, we focus on basic feature diagrams as we described before.

To construct feature models, *feature model editors* may be used. These editors provide a graphical representation of the edited feature model and allow SPL engineers to perform several modeling operations, such as adding, removing or updating features and cross-tree constraints. In Section 3.2, we discuss these operations in more detail and adapt them for use in a collaborative feature model editor.

2.1.2 Feature Model Analyses

In the literature, a number of feature model analyses have been proposed [ABKS13, BSRC10, FBGR13]. A *feature model analysis* inspects the variability expressed in a feature model to detect anomalies and inconsistencies in an SPL. Using the analysis results, SPL engineers can gain a better understanding of an SPL's variability. In the context of feature model editors, feature model analyses can be utilized to aid engineers in their modeling decisions to avoid the introduction of undesirable anomalies. We explain some common analyses, which are also employed in feature model editors:

- **Void Feature Model**. A feature model is *void* if it does not have any valid products. This hints at a faulty modeling decision because such a feature model contradicts itself, which is usually not intended by engineers.
- **Dead Features**. A feature is *dead* if it is not used in any product. Thus, it models unused potential for variability, which is usually not intended.
- **False-Optional Features**. A feature is *false-optional* if it is used in every product, but nonetheless modeled as *optional* or part of an *alternative* or *or*

group. In such cases the feature model is misleading, and the concerned feature should be modeled differently.

- **Redundant Constraints**. A cross-tree constraint is *redundant* if removing it has no impact on the feature model's products. Engineers may prefer to remove such cross-tree constraints to keep the feature model small and comprehensible.

These and other feature model analyses can also be seen as consistency properties that a feature model editor should always guarantee. In Section 5.1, we show how to integrate such consistency properties with our concept for collaborative feature modeling.

2.2 Computer-Supported Cooperative Work

The research field of Computer-Supported Cooperative Work (CSCW) addresses "how collaborative activities and their coordination can be supported by means of computer systems" [CS99]. This includes "the understanding of the way people work in groups" as well as "the enabling technologies of computer networking, and associated hardware, software, services and techniques" [Wil91]. The latter is also commonly referred to as *groupware*, i.e., "computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment" [EGR91]. For example, such collaborative activities may include project and knowledge management, communication, and conferencing as well as real-time document editing. As we show in Table 2.1, the CSCW research field covers small groups of collaborators (e.g., 3–4). This is in contrast to other research areas, which focus on individuals or larger groups [Gru94].

To classify groupware systems, Johansen has proposed a matrix that considers the working context in two dimensions, which we depict in Table 2.2 [BGBG95, Joh91]. First, collaboration may happen at the same time (*synchronous*) or at different times (*asynchronous*). For example, face-to-face meetings or conference calls are considered synchronous, as the involved parties expect immediate responses from each other; while mail and filing systems (which store information for later use) are considered asynchronous. The second distinction is whether collaboration happens at the same place (*co-location*) or at different places (*remote*). For example, face-to-face meetings require that participants are co-located, in contrast to remote conference calls. When designing a groupware solution, it is helpful to classify it according to this matrix, as it can give a hint as to how the solution may be implemented. We now investigate in more detail the class of groupware systems that applies to the concept developed in this thesis, the collaborative real-time editors.

Workspace	**Research Area**
Individual	Computer-Human Interaction
Small Group	Computer-Supported Cooperative Work, Groupware
Project	Software Engineering, Office Automation
Organization	IT, Management Information Systems

Table 2.1: Research context of CSCW, adopted from Gross and Koch [GK09].

	Co-Location	Remote
Synchronous	Face-to-Face Interactions	Remote Interactions
e.g.	*Digital whiteboards* *Electronic meeting rooms*	*Video conferencing* *Collaborative real-time editors*
Asynchronous	Ongoing Tasks	Communication and Coordination
e.g.	*Group displays* *Project management*	*Email* *Version control systems*

Table 2.2: CSCW matrix, adopted from Baecker [BGBG95].

2.2.1 Collaborative Real-Time Editors

A *collaborative real-time editor* is "a system that allows several users to simultaneously edit a document without the need for physical proximity and allows them to synchronously observe each others' changes" [Pra99]. Examples for well-known collaborative real-time editors include *Google Docs*[1], *Overleaf*[2], *SubEthaEdit*[3], *Etherpad*[4] [DDIZ18], and *Apache Wave* [WW11]. In the past decades, collaborative real-time editors have been researched in different domains, such as collaborative text editing [SJZ+98], word processing [SXSC04], graphics editing [SC02] or computer-aided design [LLFW05]; also, single-user editors such as *Microsoft Word* and *PowerPoint* have been transparently adapted to support multi-user collaboration [SXS+06].

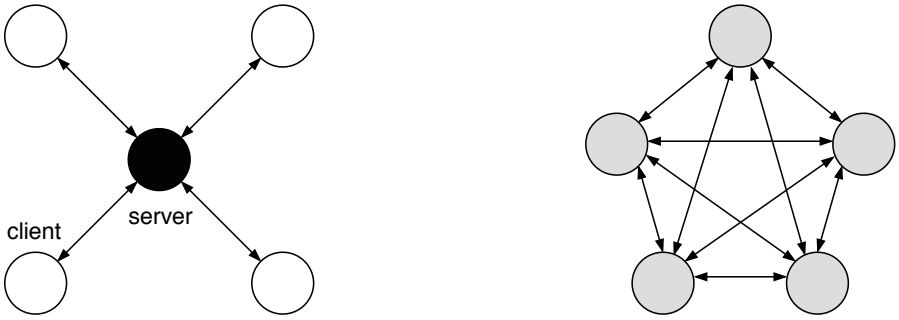
We show in Table 2.2 that collaborative real-time editors classify as synchronous and remote: They are considered synchronous because any change submitted by any collaborator is immediately propagated to every other collaborator, which provides fast and responsive feedback and gives other collaborators the opportunity for immediate review and response. Change propagation is usually done in a distributed fashion via a network (e.g., the internet) so that collaborators may participate from anywhere, which is why these editors are considered remote.

Collaborative real-time editors are more complex than single-user editors [DCS94, EG89, Pra99, SE98]. This is because they have to accommodate the needs and intentions of a number of different collaborators (which are not necessarily aligned with each other), and they have to keep collaborators updated on each other's actions. Therefore, there are a number of variation points to consider and trade-offs to make when designing a collaborative real-time editor, such as *topology*, *representation of changes*, and *level of optimism*.

Topology

Because collaborative real-time editors propagate changes via a network, its structure has to be considered. In a network, the participating collaborators (or *sites*) can connect and exchange data in different ways, referred to as topologies [Sch01].

[1] https://docs.google.com/
[2] https://overleaf.com/
[3] https://subethaedit.net/
[4] https://etherpad.org/



(a) Client/Server Architecture (b) Peer-to-Peer Architecture

Figure 2.3: Network topologies for collaborative real-time editors.

Further, we refer to the set of all sites participating in a collaboration session as the *system*. For collaborative real-time editing, mostly two topologies for the system are considered in the literature [EG89, NCDL95]: In a *client/server* architecture (or star topology, shown in Figure 2.3a) all collaborators (referred to as *client sites*) connect to a single distinguished node, the *server site*, which is responsible for registering clients and coordinating actions on the edited document. This has the benefit of efficient synchronization because a site only needs to relay changes to the server, not to all other sites; further, it allows central management of edited documents. However, the server might become a bottleneck if many clients are connected or issue editing commands. The other topology is the Peer-to-Peer (P2P) architecture we show in Figure 2.3b, where all sites are equal and communicate with some or all other sites. As there is no server present in this topology, bottlenecks are less likely; however, synchronization is more complex [SE98]. The topology of the system is important to consider because some concurrency control techniques may require specific topologies (for example, the *locking* technique we discuss in Section 4.2 usually requires a central authority, such as a server). The concept developed in this thesis may be applied in both topologies, although we focus on a client/server approach in our prototype (cf. Chapter 6).

Representation of Changes

There are two different ways for collaborative real-time editors to represent changes (or *edits*) to documents, *state-* or *operation-based* [SPBZ11b]. In the state-based approach, when a change is submitted the entire document (with the change applied) is transferred over the network channel. This is simple to implement, but has various drawbacks: For large documents, transmitting the entire document can be expensive. It is also redundant because edits usually leave most parts of a document unchanged. Further, it complicates consistency maintenance. For example, a group of users might edit a document together by always attaching their document's latest version to an email. Without manual coordination, users are then likely to create diverging documents and have to rely on manual merging.

Another approach, which we adopt in our concept, is to use *operations*. An *operation* states an action that is to be performed on the edited document, producing a new (modified) document. A user, having a specific intention in mind, issues an operation in the editor that should put this intention into practice. For example, if a user's

intention is to insert a sentence into a text document at a specific location, she has to "translate" this intent into the editing operations that particular text editor supports. This translation should be as simple and intuitive as possible to maximize editing efficiency, ideally to a point where the user can just express her intents naturally.

Operations are similar to *diffs* or *patches* found in version control systems, although operations may be tailored to the editing domain, whereas many version control systems force line-based diffs. When a groupware system employs operations, every participating site is required to store a copy (or *replica*) of the edited document [Dew99]. Such a replica may then be used as a base to generate new operations as well as to apply incoming operations received from other sites. Compared with the state-based approach, using operations to represent changes has the advantage that they are usually small and can be exchanged quickly. They are also employed by concurrency control techniques (discussed in Section 4.2 and 4.3) to allow automatic merging of divergent documents when no conflicts are present.

Level of Optimism

Collaborative real-time editors can apply changes in a *pessimistic* or *optimistic* fashion [GM94, Pra99]. A pessimistic system generally assumes that changes may be rejected by other sites (e.g., because another user is accessing the document simultaneously). Thus, it waits for the other sites to acknowledge any submitted change; only then the change is actually applied to the locally edited document. While a site is waiting for a change to be acknowledged, the user interface is blocked so that the user is unable to generate any more operations. Pessimistic client/server systems are relatively easy to design, as many consistency maintenance issues are avoided up front. However, while waiting for an acknowledgment, they block the user interface and therefore seriously restrict the users' editing capabilities. This reduces the responsiveness of the user interface, which is crucial for the efficient use of a collaborative real-time editor. This even applies to environments with short network latencies (i.e., 50 ms), which are still noticeable by users and must therefore be considered [JGH07]. Greenberg and Roseman [GR99] note that "while sluggishness is annoying when others' actions are delayed, it is devastating when the system is unresponsive to a person's own local actions, especially in highly interactive applications." In addition, all users have to be connected to a central server to submit any changes, and server outages cause the whole system to freeze. Further, pessimistic systems are not suited for larger peer-to-peer networks, as any change requires an acknowledgment from every other user.

The alternative approach, which we adopt in our concept, is an *optimistic* user interface. In optimistic systems, submitting a change does not require any acknowledgments from other sites: Instead, after generating a change, it can immediately be applied locally, before propagating it to other sites. In this case, sending changes to other sites is not intended as a request for acknowledgment, but as a simple notification that a change has been applied. Sun et al. [SJZ+98] name this approach *unconstrained collaboration* as "multiple users are allowed to concurrently and freely edit any part of the document at any time". This opens up a number of synchronization problems, as sites may apply operations locally without knowledge of the other sites' current state; thus, special concurrency control algorithms are required.

However, this also allows for a responsive user interface that immediately applies any submitted changes, effectively hiding network latencies from the user. As a consequence, users can also submit changes when they are not currently connected to the other sites (e.g., when a short network outage occurs). Further, this approach does not require a server site and allows larger peer-to-peer networks, in contrast to pessimistic approaches.

In Section 4.2, we consider several concurrency control techniques that all use operations to represent changes, but differ in their supported topologies and level of optimism. Moving on, we motivate the need for concurrency control to maintain a number of consistency properties when editing a document collaboratively.

2.2.2 Consistency Maintenance

We have already mentioned the benefits of a replicated optimistic architecture for collaborative real-time editors, namely unconstrained collaboration and high responsiveness. However, there is a price to pay when adopting such a system design: The unconstrained mode of collaboration poses some challenges with regard to maintaining consistency of edited documents; in particular, the problems of *divergence*, *causality violation*, and *intention violation* [SC02, SJZ+98]. Solving these consistency problems is the major responsibility of concurrency control in a collaborative real-time editor, as it helps to "resolve conflicts between participants, and to allow them to perform tightly coupled group activities" [EG89].

We illustrate these challenges with the example we show in Figure 2.4. In the example, three sites, A, B, and C, collaborate in a peer-to-peer fashion by generating and exchanging editing operations on a document (A, B, and C generate operations O_1, O_2/O_3 and O_4, respectively). Let us assume that if a site generates an operation, it is immediately applied locally, and operations from other sites are applied unchanged when received.

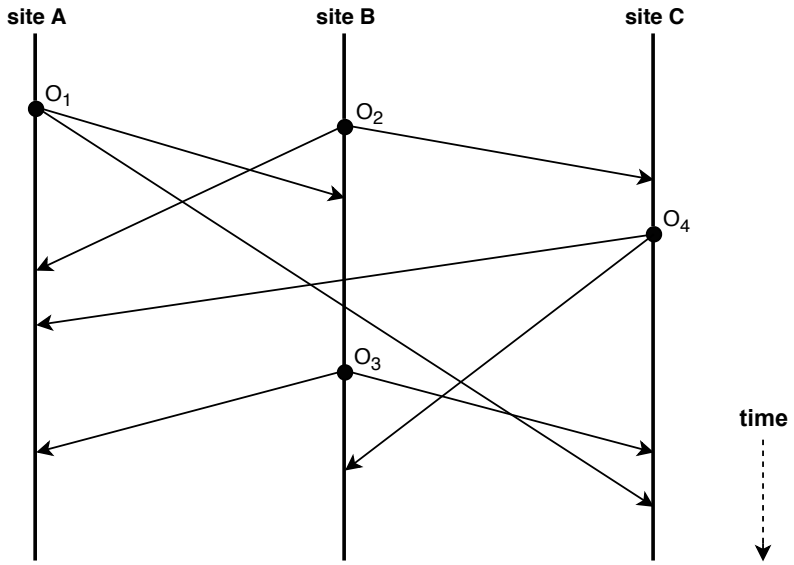


Figure 2.4: Scenario for a collaborative editing session, replicated from Sun et al. [SJZ+98]. Operations (O_i) are generated (●) and propagated to other sites.

The first problem that may occur is *divergence*: With the rules stated above, operations may be applied in different orders at different sites. For example, at site A, the four operations in the system arrive in the order O_1, O_2, O_4, and O_3; while at site B, they arrive in the order O_2, O_1, O_3, and O_4. Thus, A and B will only arrive at the same resulting document if operations in the system *commute*, i.e., yield the same resulting document regardless of execution order. Unfortunately, in most systems, such as collaborative text editing or feature modeling, editing operations are not generally commutative. Thus, site A and B will display different documents when the system is at *quiescence* (i.e., when all operations have been propagated to all sites), which is not desirable in synchronous groupware systems.

Second, network latencies may lead to *causality violation* in peer-to-peer systems. One operation may *cause* (or be *depended on* by) another. For example, to update a feature in a feature model, it must have been created previously. In general, an operation generated at a site may depend on any operations that have previously been seen at that site. In the example, the operation O_3 generated at site B (e.g., a feature edit) may refer to the execution effect of O_1, which has already arrived at site B (i.e., the creation of said feature). At site C, however, operation O_3 arrives *before* O_1 (i.e., site C is notified about the feature edit before it has been created) which may lead to an invalid document state. To avoid this effect, the system must always enforce the causal order of operations for execution.

The final problem, *intention violation*, may occur when concurrent operations are generated. Two operations are *concurrent* if they have been generated without knowledge of each other. In the example, O_1 and O_2 are concurrent, as both were generated in a context where the other operation has not yet been received. The problem may occur when concurrent operations interact in unforeseen ways. For example, suppose both O_1 and O_2 intend to set a feature's name to the different values `featureA` and `featureB`, respectively. Assuming the execution order we show in Figure 2.4, site A will end up with the feature's name set to `featureB` because the execution effect of O_2 has overridden that of O_1; similarly, site B will arrive at `featureA`. In this particular execution order, not only do both sites arrive at different results (which is the *divergence* problem discussed above), but also the intentions of both users are violated (as the system seemingly ignored the users' own operations). While the divergence problem can be avoided by reordering operations, this is not possible for intention violations: Suppose the system solves the divergence problem by executing both operations on all sites in the order O_1, then O_2 (same for O_2, then O_1). In that case, all sites will collectively set the feature's name to `featureB`. However, this will still have violated the intention of O_1, which was to set the feature's name to `featureA`, but is not reflected in the resulting document at all.

Sun et al. [SJZ+98] refer to these three challenges as *syntactic* inconsistency problems, but they also note that (depending on the editing domain) there may be other *semantic* inconsistency problems to consider. For example, in collaborative text editing, a semantic consistency property may state that edited documents should contain no grammatical errors [Sun02]. Similarly, there are several semantic consistency properties desirable in collaborative feature modeling applications (e.g., no dead features or void feature models as explained in Section 2.1.2).

Causal Ordering of Operations

To address the consistency problems described above, we have to distinguish between causally related and concurrent operations. The dependencies of events in distributed systems are usually captured using the concept of *time* [Lam78, Mat88, SM94]. In an ideal distributed system, every event (i.e., operations in a collaborative editor) may be assigned a *timestamp*, which is the physical time at which the event has occurred; timestamps of operations would then be compared to determine dependencies. However, real distributed systems suffer from clock synchronization issues, so that physical time cannot be used reliably. Further, physical timestamps are not sufficient to determine whether two operations are concurrent (because one timestamp will always precede the other). Instead, we utilize a causal ordering relation that captures all dependencies between operations. This relation was first introduced by Lamport [Lam78] and then adopted by Sun et al. [SJZ+98] for collaborative real-time editors.

Definition 2.1 (Causally-Preceding Relation [SJZ+98])**.** *Given two operations O_a and O_b, generated at sites i and j, then $O_a \rightarrow O_b$ (O_a* causally precedes *O_b) if and only if at least one of the following conditions is true:*

- *$i = j$ and the generation of O_a* happened before *the generation of O_b*
- *$i \neq j$ and the execution of O_a at site j* happened before *the generation of O_b*
- *there exists an operation O_x such that $O_a \rightarrow O_x$ and $O_x \rightarrow O_b$.*

O_a is then said to cause *O_b, while O_b* depends 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 $O_a \parallel O_b$.*

The causally-preceding relation is well-defined for two given operations and all sites can independently determine it for any two operations with a suitable mechanism (cf. Chapter 6). The relation replaces physical timestamps in the communication between different sites; nonetheless, at the same site, physical timestamps may still be used to determine whether one operation *happened before* the other. Note that we make no distinction between *time* and *causality*: If $O_a \rightarrow O_b$, we say that O_a *causes* O_b. This is a convention in distributed systems to capture all events that may have caused an event, which facilitates concurrency control [Mat88].

With the causally-preceding relation, we can capture the operation relationships depicted in Figure 2.4 as $O_1 \parallel O_2$, $O_1 \rightarrow O_3$, $O_1 \parallel O_4$, $O_2 \rightarrow O_3$, $O_2 \rightarrow O_4$, and $O_3 \parallel O_4$. We use the relation to formally specify consistency properties below and for designing the conflict detection algorithm in Section 5.1.

The CCI Model

Using the concept of causally-ordered operations, we define more precisely what properties an optimistic collaborative real-time editor has to satisfy. Guided by the problems identified above, a consistency model for optimistic editors has been proposed [SJZ+98, SYZC96]. This can be considered the standard model for optimistic concurrency in operation-based collaborative editors [SE98]. To define the model, we first specify the *intention* of an operation. We also give an informal definition of conflicts, which we revisit in Section 5.1.

Definition 2.2 (Conflicting Operations [SJZ+98])**.** *The* intention *of an operation O is the execution effect which can be achieved by applying O on the document state from which O was generated (its* generation context*). Two concurrent operations are* in conflict *if they have irreconcilable intentions (i.e., one cannot be applied without violating the intention of the other) or if together they influence the feature model in unexpected ways (such as introducing a dead feature).*

As noted by Sun et al. [SJZ+98], the intention of an operation only refers to the *syntactic effect* of an operation (e.g., inserting a character into a document), not necessarily a collaborator's semantic intention (e.g., inserting a particular sentence), which is hard to capture using operations only. The actual consistency model is then defined as follows:

Definition 2.3 (CCI Model [SJZ+98])**.** *An optimistic collaborative editing system is said to be* consistent *if it always maintains the following properties:*

- Convergence: *When the same set of operations has 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.*

Convergence guarantees that all collaborators eventually arrive at the same document (similar to strong eventual consistency in databases). In the course of an editing session, *causality preservation* is responsible for the correct ordering of dependent operations. Finally, *intention preservation* guarantees that an operation always has the same execution effect at every site regardless of its execution context (namely the one in its generation context). This entails that unexpected interactions of concurrent operations are avoided. Together, these three properties define what guarantees collaborators can expect from the system. Further, they guide the design of suitable concurrency control algorithms and allow formal reasoning on the correctness of such algorithms. In Chapter 4, we discuss which techniques are suitable to guarantee these properties in the context of collaborative feature modeling.

2.3 Related Research Areas

In this section, we discuss other well-known research areas that have their own approaches for concurrency control. Although not directly applicable, some ideas from these research areas have inspired the design of techniques specifically tailored to collaborative real-time editors [Che01].

In *database systems*, the consistency maintenance problem has primarily been investigated in the context of transaction processing [BHG87, WV02]. *Transactions* in database systems are similar to operations in collaborative real-time editors in some ways: Both describe changes to the underlying database or document that are

applied atomically (completely or not at all). Further, both are expected to preserve consistency, i.e., transition from consistent data to new consistent data again. However, transactions in database systems are often executed in an interleaved manner for improved performance. This is beneficial in database systems, which require high efficiency to process many requests and transactions simultaneously. Collaborative real-time editors, on the other hand, do not have such high performance requirements, as long as operation processing does not cause any noticeable delay for the users. Further, simultaneous processing of operations is not the regular case (as in database systems), therefore operations can simply be executed serially without negative impact on the performance. Thus, most concepts from transaction processing (such as serializability) do not directly apply to collaborative real-time editors [DCS94, EG89].

Distributed shared memory systems are similar to operation-based collaborative editors in that they also keep a replica of the edited memory or document [NL91, Che01]. Both kinds of systems strive to reduce network load by only transferring changes to the data, not the data itself. Distributed shared memory systems allow different consistency models such as *eventual consistency* (also used in database systems), where the participating sites converge towards a globally equal end result, which is similar to the *convergence* property introduced in the previous section. However, distributed shared memory systems do not address the specific issues of operations and intention preservation required for collaborative real-time editors.

Version control systems are another common means to collaborate on documents and projects [Cha09, O'S09, PCSF08]. They can be classified as asynchronous remote groupware according to Table 2.2. Version control systems share some similarities with operation-based collaborative real-time editors: Both allow remote editing of shared documents and provide facilities for synchronization and resolving emerged conflicts. Further, both use "deltas" (diffs and operations, respectively) to represent changes. However, version control systems are intended for *asynchronous* collaboration. This refers to scenarios where multiple collaborators work on different parts of the edited document or project mostly independently, and only synchronize their changes manually and sporadically. In contrast, collaborative real-time editors employ a synchronous mode of operation where synchronization occurs frequently, and immediate feedback from other collaborators is encouraged and expected.

For example, when editing source code collaboratively, the use of version control systems promotes a rather separated way of working. This is because collaborators seek to avoid merge conflicts if possible. For scenarios in which tight collaboration is necessary and even desired to leverage synergy effects (such as pair programming [GLM11]), collaborative real-time editors are more suitable. Fraser [Fra09] draws the following analogy: "[A version control system] could be compared to an automobile with a windshield which becomes opaque while driving. Look at the road ahead, then drive blindly for a bit, then stop and look again. Major collisions become commonplace when everyone else on the road has the same type of 'look xor drive' cars."

On the technical side, version control systems commonly represent changes with *line-based diffs*, which do not easily translate to tree-based feature models [SW17]. More importantly, version control systems usually have a notion of conflicts where all

changes that affect the same region of the edited document are considered conflicting. Because features can have complex relationships with each other, this definition of conflict is too narrow to ensure higher levels of semantic consistency (which depend on these relationships). Nevertheless, some concepts from version control systems, such as branching and merging, are indeed useful for collaborative feature modeling (cf. Section 4.3).

2.4 Summary

In this chapter, we provided background knowledge required to build a collaborative real-time feature model editor. We settled on the *basic feature diagram* notation for editing feature models and introduced a number of feature model consistency properties. Then, we examined collaborative real-time editors and discussed advantages of operation-based optimistic systems. We also introduced fundamental concepts from consistency maintenance, which we use to design our concept in Section 5.1. Finally, we discussed related research areas and why their approaches for concurrency control do not suffice for collaborative real-time editors.

3. Feature Models & Operations

All editing software relies on suitable document and operation models to provide end users with proper editing capabilities. The *document model* of an editor determines the basic structure and representation of the edited documents, while the *operation model* specifies which manipulations on documents are available to the user. In collaborative real-time editors, document and operation models must be designed carefully to support suitable consistency maintenance mechanisms, as this may impact the editing experience of collaborators. In this chapter, we devise both a document and an operation model for collaborative real-time feature modeling. First, we discuss and formalize how we represent feature models in a way that facilitates later concurrency control. Then, we review feature modeling operations in existing feature model editors and introduce operations suitable for multi-user collaboration.

3.1 Representing Feature Models

To develop a collaborative feature model editor, we have to identify a suitable feature model representation. There are two major variation points for the definition of feature models (already discussed briefly in Section 2.1.1): First, notions of feature models may differ in the kinds of variability which can be modeled (e.g., we may choose to add arbitrary attributes or cardinalities) [BTRC05, CHE04]. Second, feature models have different representations, such as feature diagrams or propositional formulas [Bat05]. Choosing a concrete feature model definition and representation impacts all following layers in our concept's architecture, i.e., operations and possible conflicts.

For this thesis, we choose to use *basic feature models* represented as *feature diagrams* (cf. Section 2.1.1), i.e., *basic feature diagrams*. We opt for this approach for several reasons: First, basic feature diagrams are a well-known and human-readable means to model variability with established tool support [BRN+13, MTS+17]. This makes them well-suited for multi-user adaptation, because collaborators with prior feature modeling experience may simply use accustomed modeling techniques. Second, they

serve as a solid base for other feature model definitions and representations. For example, *extended feature models* are similar to basic feature models, only adding feature properties; also, feature diagrams are simple to convert into other representations (such as propositional formulas). Third, basic feature diagrams already demonstrate well a number of issues that arise in collaborative feature modeling. For example, despite their simple tree structure, they impose a number of conditions on valid feature models that need to be maintained while editing (discussed below). Thus, our concept, while focusing on basic feature diagrams, may be extended to other definitions and representations in the future.

We proceed with a formalization of feature models as basic feature diagrams. We use this formalization throughout the thesis to provide an accurate description of our concept and its properties. It is specifically designed to facilitate collaborative feature modeling and therefore extends previous formalizations (cf. [SHT06]).

3.1.1 Syntax of Feature Models

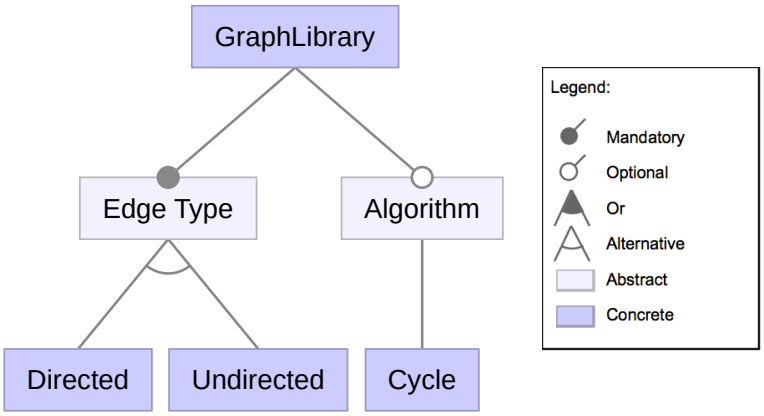
We begin with defining the simple, but important set of identifiers in the system.

Definition 3.1. *The infinite set $\mathcal{ID}$ contains all Universally Unique Identifiers (UUIDs) that may potentially be used by the system at some point.*

The purpose of $\mathcal{ID}$ is to be able to assign a unique identifier (ID for short) to every entity in the system (i.e., features, cross-tree constraints, and operations) when it is created. These identifiers are then used throughout the system to generate and apply operations and also to detect conflicts. This is a common practice in object-based (e.g., Computer-Aided Design (CAD) or graphics editing) systems and simplifies concurrency control [Che01, LCD+07, TCD07]. The definition is somewhat vague with respect to the nature of IDs. This is intentional, as it permits any implementation that is able to produce a new unique ID when requested. For example, elements of $\mathcal{ID}$ may be random number or string identifiers, unique for every created entity. Using $\mathcal{ID}$, we can now define the basic syntax of features, cross-tree constraints, and feature models.

Definition 3.2. *A* feature *is a tuple (ID, parentID, optional, groupType, abstract, name) where $ID \in \mathcal{ID}$, $parentID \in \mathcal{ID} \cup \{\bot, \dagger\}$, $optional \in \{true, false\}$, $groupType \in \{and, or, alternative\}$, $abstract \in \{true, false\}$, and name is a string.*
A cross-tree constraint *is a tuple $(ID, \phi, \dagger)$ where $ID \in \mathcal{ID}$, ϕ is a propositional formula with variables ranging over $\mathcal{ID}$, i.e., $Var(\phi) \subseteq \mathcal{ID}$, and $\dagger \in \{true, false\}$.*
A feature model *FM then is a tuple $(\mathcal{F}, \mathcal{C})$ where $\mathcal{F}$ is a finite set of features, and $\mathcal{C}$ is a finite set of cross-tree constraints.*

This definition captures all essential properties of basic feature diagrams as discussed in Section 2.1.1, while at the same time, it integrates well with our consistency maintenance approach introduced in Chapter 5. As mentioned, features and cross-tree constraints are assigned IDs. In addition to an identifier, feature and cross-tree constraint tuples then specify several attributes, starting with the *parentID* attribute for features. Almost every feature has a parent, which is stored using its ID, to capture the basic tree structure. $\bot$ denotes the root feature, which has no parent.



$$Cycle \Rightarrow Directed$$

Figure 3.1: An excerpt of the graph product line shown in Figure 2.2.

† is a simple removal flag for features and cross-tree constraints, which we elaborate on below.

Further, feature tuples include an attribute to express whether a feature is mandatory or optional, and another attribute that models a feature's group type. These attributes are included because they require special attention in the conflict detection. More attributes may be added; for example, we consider additional *abstract* and *name* attributes, both of which are commonly used: Abstract features are a hint for product line engineers that a feature is only used to structure the feature model. Names are used to label features in the editor's user interface (whereas IDs are not suitable as feature labels because they are unique and may not be changed after feature creation). Note that although these additional attributes may resemble attributes in extended feature models, they are different: In extended feature models, cross-tree constraints may also range over additional attributes, which we do not permit here. Instead, every cross-tree constraint is associated with a propositional formula that specifies further restrictions on the set of valid products. Similar to features, the † attribute (discussed below) is a removal flag for cross-tree constraints.

Example 3.1. As an example, we show a basic feature diagram in Figure 3.1 and represent it according to Definition 3.2.

Let $FM = (\mathcal{F}, \mathcal{C})$ where

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

$$\text{and } \mathcal{C} = \{(constraint1, Cyc \Rightarrow Dir, false)\}.$$

FM then corresponds to the basic feature diagram shown in Figure 3.1. The IDs in this example are arbitrary and will, in practice, be generated automatically by the system. The `GPL` feature is the root feature because its parent ID is $\perp$. Every other feature has a *parentID* that identifies its parent in the feature tree. The underlined attribute values are effectively ignored by the user interface: For instance, the `Directed` feature is part of an *alternative* group, which overrides its optional flag. For such features, the feature diagram does not include a mandatory/optional circle.

3.1.2 Legal Feature Models

Definition 3.2 so far captures all basic feature diagrams, but it also allows a number of "feature models" that violate integral properties of basic feature diagrams. In particular, Definition 3.2 does not require that feature IDs are unique, or that referenced feature IDs are actually defined by the feature tree, so feature parents and cross-tree constraints may refer to "dangling" features. In addition, the current definition only constrains the "feature tree" to a graph where every feature has in-degree zero or one. To guarantee that the feature tree is actually a tree, it is also required to have a single root and no cycles. To ensure these additional properties, we define what it means for a feature model to be *legal*.

For better readability, we introduce a dot notation to access a tuple's attributes, e.g., we may write $FM.\mathcal{F}$ to denote the set of features in the feature model FM. In addition, we interpret the $\mathcal{F}$ and $\mathcal{C}$ sets as functions to allow easy look-up of features and cross-tree constraints for a given unique identifier. For example, in the context of Example 3.1 we may write $FM.\mathcal{F}(Dir).parentID$ to denote the value ET of the *parentID* field for the feature identified by Dir.

Definition 3.3. *Let FM be a feature model. Further, define*

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

FM is then considered legal *if and only if all of the following conditions are true:*

- **Unique Identifiers***:* $|FM.\mathcal{F}| = \left|\mathcal{F}_{FM}^{ID}\right|$ *and* $|FM.\mathcal{C}| = \left|\mathcal{C}_{FM}^{ID}\right|$
- **Valid Parents***: for all features* $F \in FM.\mathcal{F}$*,* $F.parentID \in \mathcal{F}_{FM}^{ID} \cup \{\perp, \dagger\}$
- **Valid Constraints***: for all constraints* $C \in FM.\mathcal{C}$*,* $Var(C.\phi) \subseteq \mathcal{F}_{FM}^{ID}$
- **Single Root***:* $F.parentID = \perp$ *for exactly one feature* $F \in FM.\mathcal{F}$
- **Acyclic***: for all feature IDs* $F^{ID} \in \mathcal{F}_{FM}^{ID}$*,* $F^{ID} \preceq_{FM} \perp$ *or* $F^{ID} \preceq_{FM} \dagger$*.*

The set of all legal feature models is denoted as $\mathcal{FM}$*.*

Example 3.1 fulfills all stated conditions and is therefore a legal feature model. One responsibility of our concurrency control approach is to ensure that edited feature models are always legal.

3.1.3 Removing Features and Cross-Tree Constraints

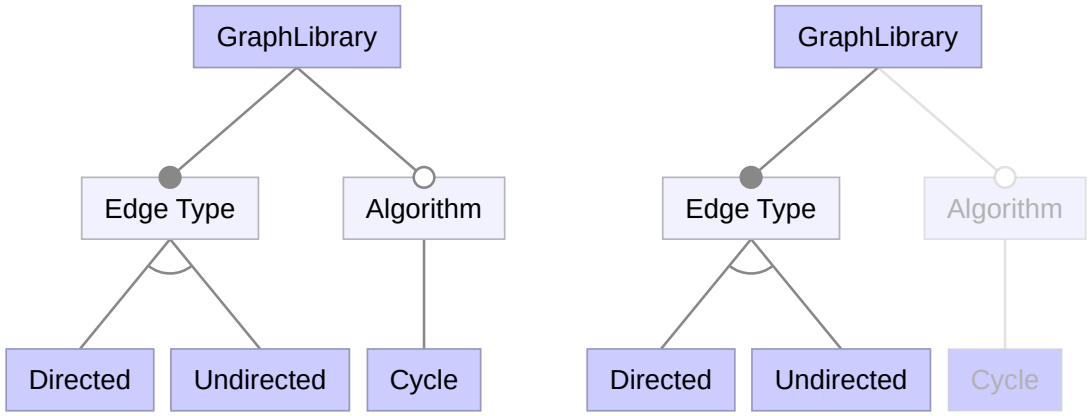
To explain how we handle deletion of features and cross-tree constraints, we consider an interaction between deletion operations and undo/redo (a facility in many editors that lets users reverse their changes): Suppose a collaborator requests to delete a feature, but later on changes her mind and undoes the deletion. The default approach would be to let a deletion operation remove all traces of the deleted feature from the model, i.e., remove the feature from $FM.\mathcal{F}$. However, with this approach it is impossible to undo a deletion operation because the deleted feature is no longer available in the system. Instead, deleted features (and cross-tree constraints) should be kept in the system, but marked as removed. In established distributed concurrency control techniques, the removal mark on an object is known as a *tombstone* [CS01b, OMUI06, SPBZ11b]. Following this practice, we introduce a so-called *graveyard* that comprises removed features and cross-tree constraints. The distinguished feature *parentID* † signifies that a feature has been removed. In fact, † can be seen as a second root that spans the tree of removed features. This way, we can not only undo removal operations, but also represent them as *move* operations, which facilitates conflict detection in Section 5.1. As for cross-tree constraints, † is a simple removal flag that is *false* by default and *true* when a cross-tree constraint has been removed explicitly.

We now define precisely when features and cross-tree constraints are *graveyarded*. A graveyarded feature or cross-tree constraint will not be displayed in the user interface unless the removal operation is undone.

Definition 3.4. *Let $FM \in \mathcal{FM}$ be a legal feature model. A feature $F \in FM.\mathcal{F}$ is* graveyarded *if and only if $F.ID \preceq_{FM} \dagger$. Further, a cross-tree constraint $C \in FM.\mathcal{C}$ is* graveyarded *if and only if $C.\dagger = true$ or any feature identified by an ID in $Var(C.\phi)$ is graveyarded. The* feature graveyard $\mathcal{F}^{\dagger}_{FM}$ *is then the set of all identifiers of graveyarded features. Respectively, the* cross-tree constraint graveyard $\mathcal{C}^{\dagger}_{FM}$ *is the set of all identifiers of graveyarded cross-tree constraints.*

There are two things worth noting here: First, because $\preceq_{FM}$ is transitive, a feature is also considered graveyarded if any of its ancestors has been removed. This means that, by default, entire feature subtrees are removed; and that the subtrees' internal structure is preserved on the graveyard. We discuss this further when designing the removal operations in Section 3.2.

Second, cross-tree constraints may not only be graveyarded by collaborators explicitly, but also implicitly if any referenced feature is graveyarded. This means that if a feature is removed, all cross-tree constraints concerning that feature are also removed. This is necessary because otherwise, these cross-tree constraints would still be visible, but refer to a removed feature. (Another approach here may be to entirely forbid removing features with associated cross-tree constraints. This is trivial to implement by rejecting such removal operations immediately.)



(a) Initial feature model

(b) Removing a feature subtree

Figure 3.2: Performing a removal operation in a feature model. Graveyarded objects are faded out.

Example 3.2. We give an integrated example to demonstrate the graveyard as defined in Definition 3.4. Suppose a collaborator is editing the feature model shown in Figure 3.2a. In the beginning, no features or cross-tree constraints are graveyarded, so $FM = (\mathcal{F}, \mathcal{C})$ where

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

Initially, $\mathcal{F}^{\dagger}_{FM} = \mathcal{C}^{\dagger}_{FM} = \varnothing$. Now, the collaborator decides to remove the subtree formed by the `Algorithm` feature, so she expects that `Algorithm` and `Cycle` are removed. Not considering for now how the removal operation actually operates, this will modify `Algorithm`'s feature tuple as follows, shown in Figure 3.2b:

$$(Alg, \dagger, true, or, true, \texttt{Algorithm})$$

Note that the features `Algorithm` and `Cycle` are still present in the feature tree, but `Algorithm` is marked as removed by setting its parent ID to †. By the semantics given in Definition 3.4, this will cause the graveyards to be $\mathcal{F}^{\dagger}_{FM} = \{Alg, Cyc\}$ and $\mathcal{C}^{\dagger}_{FM} = \{constraint1\}$. `Algorithm` is included because it directly descends from †, while `Cycle` is included indirectly because its parent feature `Algorithm` is graveyarded. Because it refers to the now graveyarded `Cycle`, the only cross-tree constraint is implicitly graveyarded as well (not explicitly, as the † attribute of *constraint1* is still false).

Suppose the collaborator later on decides that this change should be undone, so she generates an undo operation and expects this to restore the `Algorithm` subtree. Such an undo operation will simply set $FM.\mathcal{F}(Alg).parentID$ back to GPL. After this operation, the feature model will be back to the one we started with, and $\mathcal{F}^{\dagger}_{FM} = \mathcal{C}^{\dagger}_{FM} = \varnothing$. Note that in this case, the cross-tree constraint has been restored as well. This is not necessarily the case, as other collaborators may have removed the feature `Directed` in the meantime, which would cause the cross-tree constraint to be graveyarded nonetheless. In that case, we may notify the involved collaborators about the situation to avoid surprises.

Example 3.2 shows the key benefits of using $\mathcal{F}^{\dagger}_{FM}$ and $\mathcal{C}^{\dagger}_{FM}$ to store removed features and cross-tree constraints: First, it preserves the structure of removed feature subtrees, so that removing and restoring them only requires setting one attribute (the parent ID) of one feature. Second, it enables undo/redo of removal operations on the feature model.

3.2 Designing an Operation Model

In this section, we discuss the importance and nature of operations in single-user editors. We then review feature modeling operations in existing single-user editors. Finally, we design an operation model suitable for collaborative real-time feature modeling and justify the correctness of our approach.

3.2.1 Operations in Single-User Editors

Referring to editor software in general, operations are a user's primary means to manipulate a document. In Section 2.2.1, we already stated that users should be able to intuitively translate their intentions to concrete editing operations supported by the editor. In the text editing domain, the basic concept of a *cursor* that marks the user's current insertion location has prevailed [EG89, GM94, RNRG96]. For example, a user might insert a sentence into a text document by positioning her cursor at the specific location she has in mind, and then typing the sentence character by character.

Graphics editors, on the other hand, follow a slightly different approach. In graphical editing systems (which include drawing, designing and modeling activities), usually there is no linear (i.e., position-indexed) data model that may be addressed using a cursor. Instead, most graphics editors structure a document either as a *bitmap*, i.e., a two-dimensional grid of pixels, or as an unsorted collection of *objects* (and their *relationships*, if any) [Che01]. In an object-based graphics editor, an *object* might be a graphical shape or a class in a class diagram. Relationships between objects depend on the concrete application: For example, in drawing applications there often is the concept of *layers* to determine which object is in the foreground; while in many modeling applications, two objects can be linked (e.g., class inheritance or feature-parent-relationship). Instead of addressing objects implicitly by their position in the document, graphics editors typically assign unique identifiers to objects [Che01, LCD+07, TCD07]. Operations are then said to *target* one or multiple objects using these identifiers. In the user interface, this is often communicated to a user with

a *selection*, i.e., the user first creates a selection including all objects the operation should target, then proceeds with the actual operation.

Clearly, feature model editors fall into the range of graphics editors, where features (or possibly, entire subtrees of features) are considered objects. Relationships between features are then expressed through the basic tree structure of a feature diagram, together with additional cross-tree constraints. In Section 3.1, we defined feature models in a way that fits this operation model, as each feature and cross-tree constraint is assigned a unique identifier which can serve as an operation target. Because the $\mathcal{ID}$ set provides infinite identifiers and a feature model always has a finite number of features and cross-tree constraints, we can always obtain such a unique identifier.

3.2.2 Single-User Feature Modeling Operations

We have reviewed two well-known feature modeling tools, *FeatureIDE* [MTS+17] and *pure::variants* [Beu08], to identify feature modeling operations which guide our design. This has the benefit that users that are acquainted with single-user feature modeling tools have little trouble adapting to collaborative usage; also, it facilitates integration of our concept with existing tools.

In FeatureIDE and pure::variants, operations that affect the feature model can be grouped into three categories: view-, feature- and constraint-related. First, *view-related operations* serve to adjust the visual representation of a feature model to a user's needs. For example, both tools support a number of layouts, including tree, table, and graph visualizations. As feature models may become quite large, feature subtrees may be collapsed and expanded to keep track of relevant features and cross-tree constraints. We argue that view-related operations do not have to be considered for collaborative feature modeling, as they do not have any effect on the semantics of the edited feature model. This permits collaborators to use any view that best serves their preferences and intentions (e.g., they may collapse different feature subtrees independently). Thus, view-related operations are only applied locally and do not contribute to collaborative activities.

Second, *feature-related operations* include creation, removal and updates of features in the feature tree. For feature creation, FeatureIDE and pure::variants allow the user to create a feature below an existing parent feature. FeatureIDE additionally supports adding a feature above or as sibling to other features. Both tools also allow moving a feature below another feature, as well as reordering sibling features. For removing features, both FeatureIDE and pure::variants allow the user to remove an entire feature subtree (i.e., a feature and all its children). FeatureIDE can also remove a single feature, pulling up child features in the process. Finally, both tools allow updating various properties of features, including the *optional* and *group type* attributes discussed in Section 3.1. Both also allow the user to assign a human-readable name and description to a feature. FeatureIDE additionally allows setting a number of properties that are only used for improving the structure and readability of a feature model, namely, the *abstract*, *hidden*, and *color* attributes.

Third, both tools include support for arbitrary cross-tree constraints and therefore *constraint-related operations*. Cross-tree constraints can be created, edited, removed,

and reordered in both tools. pure::variants provides different predefined types of relationships (e.g., *requires* and *conflicts*), but also allows arbitrary cross-tree constraints expressed in *Prolog* or the expression language *pvSCL*.

Further, operations in FeatureIDE and pure::variants carry a target identifier that specifies the affected feature or cross-tree constraint. Both tools also offer *multi-target operations* (such as *remove*) that operate on a selection of multiple features or cross-tree constraints. FeatureIDE identifies features by name, while pure::variants assigns randomly generated unique identifiers. In a collaborative context, identification by name is discouraged because features may be renamed, which complicates concurrency control [Che01]. Thus, we adopt an approach similar to pure::variants with unique identifiers from the $\mathcal{ID}$ set introduced in Section 3.1, while representing feature names with an additional *name* attribute.

3.2.3 Collaborative Feature Modeling Operations

In the last three decades, research on operation-based concurrency control has focused on a small set of operations, namely: *Insertion* and *deletion* of objects, often in the field of text editing [Cor95a, SJZ+98], extended by *update* operations in graphics editors [SC02, SXSC04]. Systems that use such a small set of "core operations" are advantageous because they can be built and reasoned about more easily than systems that reconcile consistency of many different types of operations. In fact, even when restricting oneself to only inserting and deleting characters in a text document, it is difficult to design an optimistic system that achieves the CCI properties discussed in Section 2.2.2 [IROM06, SXA14]. Nonetheless, a collaborative feature model editor has to support a wide variety of operations to compete with single-user editors such as FeatureIDE and pure::variants.

To be able to build on existing research and reason about correctness while having all the advantages of a fully-fledged feature model editor, we have decided on a two-layered operation architecture where we distinguish two kinds of operations (inspired by *primitive* and *adapted* operations in [SXS+06]): On the lower level, Primitive Operations (POs) represent fine-grained edits to feature models and are suitable for applying concurrency control techniques from prior research (discussed in Section 4.3). These operations are simple and composable, which facilitates reasoning about correctness and later extension. On the higher level, Compound Operations (COs) expose actual feature modeling operations to the application. A compound operation is simply a sequence of primitive operations that should be executed in order. All feature modeling operations discussed above can be expressed as compound operations.

Using this two-layer architecture instead of a single large set of operations has a number of advantages: First, when detecting conflicts between operations, we can focus on primitive operations and do not have to treat compound operations specially, as they are simple PO sequences. For the same reason, multi-target operations discussed above such as *remove* are simple to implement by composing several single-target COs into one new, large CO. Also, to extend this design with new functionalities, it is sufficient to design new compound operations, without needing to make major changes to the conflict detection. We proceed to give an overview of the initial primitive and compound operations in our concept.

3.2.4 Primitive Operations

Single-user feature modeling tools allow creating, removing and updating features and cross-tree constraints in various ways. We now present five primitive operations that serve as building blocks for these larger operations. A Primitive Operation (PO) is a collection of attributes that represent a change to a feature model. In this section, we can distinguish three categories of operations: *create*, *update*, and *assertion* POs. For each PO, we list its attributes and a short description. Further, we provide formal semantics using the feature model formalization presented in Section 3.1. These semantics are expressed in form of pre- and postconditions. The former define the circumstances when a PO may be applied, while the latter define a PO's application effect (the actual implementation may vary). In the pre- and postconditions, FM and FM' refer to the feature model before and after PO application, respectively.

First, we introduce two operations to create and update features in the feature tree.

createFeaturePO(F^{ID})

Creates a feature with a globally new 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

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

postconditions

$F^{ID} \in \mathcal{F}^{ID}_{FM'}$
$FM'.\mathcal{F}(F^{ID}).parentID = \dagger$
$FM'.\mathcal{F}(F^{ID}).optional = true$
$FM'.\mathcal{F}(F^{ID}).groupType = and$
$FM'.\mathcal{F}(F^{ID}).abstract = false$
$FM'.\mathcal{F}(F^{ID}).name =$ `New Feature`

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

Updates a single attribute of a single feature to a new value. The feature is addressed with an existing ID. The old attribute value is also included when the operation is generated, this is required for conflict detection. *Dom* refers to the attribute's domain as defined in Section 3.1, e.g., $Dom(parentID) = \mathcal{ID} \cup \{\bot, \dagger\}$. Further, $FM.\mathcal{F}(F^{ID}).[attribute]$ refers to a particular feature attribute value as specified by *attribute*.

preconditions

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

postconditions

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

For example, creating a feature *Weighted* with default attributes and inserting it below the feature *GraphLibrary* may be expressed with two primitive operations: *createFeaturePO*(*Weighted*), followed by *updateFeaturePO*(*Weighted*, *parentID*, †, *GraphLibrary*).

Further, we define analogous operations for creating and updating cross-tree constraints and their attributes.

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**
$C^{ID} \in \mathcal{ID}$	$C^{ID} \in \mathcal{C}^{ID}_{FM'}$
$C^{ID} \notin \mathcal{C}^{ID}_{FM}$	$FM'.\mathcal{C}(C^{ID}).\phi = \top$
	$FM'.\mathcal{C}(C^{ID}).\dagger = true$

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**
$C^{ID} \in \mathcal{C}^{ID}_{FM}$	$FM'.\mathcal{C}(C^{ID}).[attribute] = newValue$
$attribute \in \{\phi, \dagger\}$	
$oldValue = FM.\mathcal{C}(C^{ID}).[attribute]$	
$newValue \in Dom(attribute)$	

As discussed in Section 3.1, dedicated *removal* operations are not required, because they can be expressed as *update* operations setting either the *parentID* to † (for features) or † to *true* (for cross-tree constraints).

Finally, we define a single *assertion* PO, which is needed to detect conflicts related to feature removal operations in Section 5.1.

assertNoChildAddedPO(F^{ID})

Used as a hint to the conflict detection that concurrent operations may add no children to a feature. Has no execution effect on a feature model.

preconditions	**postconditions**
$F^{ID} \in \mathcal{F}^{ID}_{FM}$	—

These five operations can be used to assemble complex feature modeling operations.

3.2.5 Compound Operations

The above-defined primitive operations should not be exposed as is to a feature modeling application because in general, they cannot be directly mapped to operations issued by users (which typically set multiple attributes on different features or cross-tree constraints). Therefore we represent user-issued operations as sequences of primitive operations that are applied atomically, referred to as Compound Operations (COs).

We now define the initial compound operations in our concept. Similar to the primitive operations, we give a short description and preconditions for each CO. As compound operations are sequences of POs, we further provide an algorithm for each CO that generates said PO sequence. Whenever the user requests to generate a CO, the required preconditions are checked with regard to the current feature model (FM), then the associated algorithm is invoked with FM and any required arguments, e.g., a feature parent (FP) or feature children (FC). The generated CO can then be applied locally and subsequently transmitted to all other collaborators.

First, we discuss feature-related operations, introducing compound operations for creating and inserting features into the feature tree.

createFeatureBelow creates a feature F^{ID} below another feature FP^{ID}.
function CREATEFEATUREBELOW(FM, F^{ID}, FP^{ID})
Require: $F^{ID} \in \mathcal{ID}$, $F^{ID} \notin \mathcal{F}^{ID}_{FM}$, $FP^{ID} \in \mathcal{F}^{ID}_{FM}$
  return [$createFeaturePO(F^{ID})$, ▷ create the new feature
    $updateFeaturePO(F^{ID}, parentID, \dagger, FP^{ID})$] ▷ insert it into the tree
end function

createFeatureBelow suffices to express all feature model edits involving feature creation, but for some editing intentions (such as creating a feature above the root feature) a dedicated *createFeatureAbove* operation is more convenient.

createFeatureAbove creates a feature F^{ID} above a set of sibling features FC^{IDs}. Commonly, the feature is created above only one feature ($\left|FC^{IDs}\right| = 1$).
function CREATEFEATUREABOVE(FM, F^{ID}, FC^{IDs})
Require: $F^{ID} \in \mathcal{ID}$, $F^{ID} \notin \mathcal{F}^{ID}_{FM}$, $FC^{IDs} \neq \varnothing$, $\forall FC^{ID} \in FC^{IDs} : FC^{ID} \in \mathcal{F}^{ID}_{FM}$,
    ▷ the features in FC^{IDs} must be siblings
    $\forall FC^{ID}_a, FC^{ID}_b \in FC^{IDs} : FM.\mathcal{F}(FC^{ID}_a).parentID = FM.\mathcal{F}(FC^{ID}_b).parentID$
  $CO \leftarrow [\,]$
  $FP^{ID} \leftarrow FM.\mathcal{F}(FC^{ID}).parentID$ for any $FC^{ID} \in FC^{IDs}$
  $CO.append(createFeaturePO(F^{ID}))$ ▷ create the new feature
  $CO.append(updateFeaturePO(F^{ID}, parentID, \dagger, FP^{ID}))$ ▷ insert into tree
  if $FP^{ID} \neq \perp$ **then**
    ▷ if not creating above the root feature, adjust the group type
    $CO.append(updateFeaturePO(F^{ID}, groupType, and, FM.\mathcal{F}(FP^{ID}).groupType))$
  end if

for all $FC^{ID} \in FC^{IDs}$ **do**
▷ move the sibling features below the new feature
$CO.append(updateFeaturePO(FC^{ID}, parentID, FP^{ID}, F^{ID}))$
end for
return CO
end function

We further present operations to move and remove features in the feature tree.

moveFeatureSubtree 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.

function MOVEFEATURESUBTREE(FM, F^{ID}, FP^{ID})
Require: $F^{ID}, FP^{ID} \in \mathcal{F}^{ID}_{FM}$, $FP^{ID} \not\preceq_{FM} F^{ID}$
return $[updateFeaturePO(F^{ID}, parentID, FM.\mathcal{F}(F^{ID}).parentID, FP^{ID})]$
end function

removeFeatureSubtree 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.

function REMOVEFEATURESUBTREE(FM, F^{ID})
Require: $F^{ID} \in \mathcal{F}^{ID}_{FM}, FM.\mathcal{F}(F^{ID}).parentID \neq \bot$
return $[updateFeaturePO(F^{ID}, parentID, FM.\mathcal{F}(F^{ID}).parentID, \dagger)]$
end function

removeFeature 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, which becomes the new root feature.

function REMOVEFEATURE(FM, F^{ID})
Require: $F^{ID} \in \mathcal{F}^{ID}_{FM}, FM.\mathcal{F}(F^{ID}).parentID = \bot \Rightarrow \left|FC^{IDs}\right| = 1$
$CO \leftarrow [\,]$
$FP^{ID} \leftarrow FM.\mathcal{F}(F^{ID}).parentID$
$FC^{IDs} \leftarrow \{FC^{ID} \in \mathcal{F}^{ID}_{FM} \mid FM.\mathcal{F}(FC^{ID}).parentID = F^{ID}\}$
▷ required for conflict detection in Section 5.1
$CO.append(assertNoChildAddedPO(F^{ID}))$
for all $FC^{ID} \in FC^{IDs}$ **do**
▷ pull the feature's children up one level
$CO.append(updateFeaturePO(FC^{ID}, parentID, F^{ID}, FP^{ID}))$
end for
▷ move the feature to the graveyard
$CO.append(updateFeaturePO(F^{ID}, parentID, FP^{ID}, \dagger))$
return CO
end function

As mentioned in Section 3.1, *removeFeatureSubtree* simply sets the *parentID* of the removed subtree's root to †. This not only graveyards said root, but also every other feature in that subtree (cf. Definition 3.4). Note that no *parentIDs* of other features than the subtree root have been touched. This way, the subtree's internal structure is preserved when it is graveyarded, which simplifies undoing a *removeFeatureSubtree* operation (just restore the subtree root's original *parentID*).

For each additional feature attribute, operations are introduced to set the attribute values. Exemplary, we give the definition of *setFeatureOptional*—the definitions of the *setFeatureGroupType*, *setFeatureAbstract* and *setFeatureName* operations are analogous.

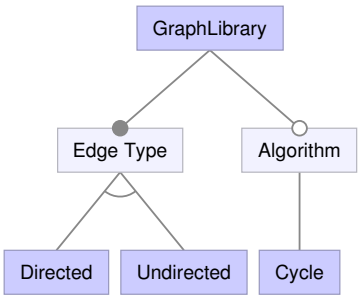
setFeatureOptional sets the *optional* attribute of a feature F^{ID} to *newValue*.
function setFeatureOptional(FM, F^{ID}, *newValue*)
Require: $F^{ID} \in \mathcal{F}_{FM}^{ID}$, $newValue \in Dom(optional)$
  return $[updateFeaturePO(F^{ID}, optional, FM.\mathcal{F}(F^{ID}).optional, newValue)]$
end function

Finally, we present constraint-related operations. Again, there are operations for creating, updating and removing cross-tree constraints. These operations are simpler than their feature-related counterparts because cross-tree constraints are not organized in a tree structure, but rather in a simple set.

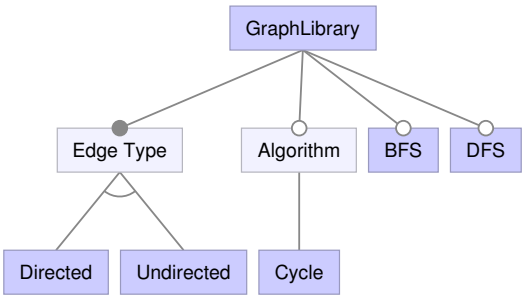
createConstraint creates a cross-tree constraint C^{ID} and initializes it with a given propositional formula ϕ_{init}.
function createConstraint(FM, C^{ID}, ϕ_{init})
Require: $C^{ID} \in \mathcal{ID}$, $C^{ID} \notin \mathcal{C}_{FM}^{ID}$, $\phi_{init} \in Dom(\phi)$, $Var(\phi_{init}) \subseteq \mathcal{F}_{FM}^{ID}$
  return $[createConstraintPO(C^{ID}),$ ▷ create the new constraint
    $updateConstraintPO(C^{ID}, \phi, \top, \phi_{init}),$ ▷ initialize ϕ
    $updateConstraintPO(C^{ID}, \dagger, true, false)]$ ▷ remove from graveyard
end function

setConstraint sets the propositional formula of a constraint C^{ID} to ϕ_{new}.
function setConstraint(FM, C^{ID}, ϕ_{new})
Require: $C^{ID} \in \mathcal{C}_{FM}^{ID}$, $\phi_{new} \in Dom(\phi)$, $Var(\phi_{new}) \subseteq \mathcal{F}_{FM}^{ID}$
  return $[updateConstraintPO(C^{ID}, \phi, FM.\mathcal{C}(C^{ID}).\phi, \phi_{new})]$
end function

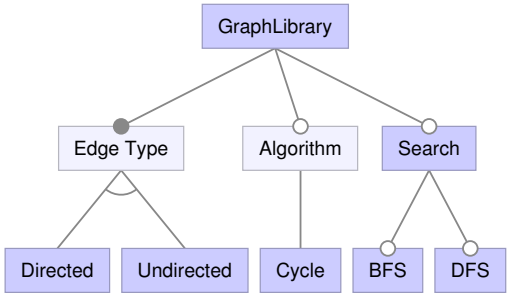
removeConstraint removes a cross-tree constraint C^{ID}. The removed cross-tree constraint is graveyarded.
function removeConstraint(FM, C^{ID})
Require: $C^{ID} \in \mathcal{C}_{FM}^{ID}$
  return $[updateConstraintPO(C^{ID}, \dagger, FM.\mathcal{C}(C^{ID}).\dagger, true)]$
end function



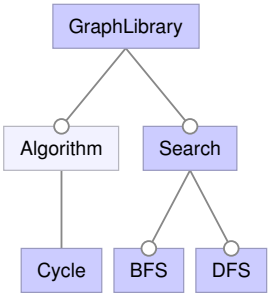
(a) Initial feature model



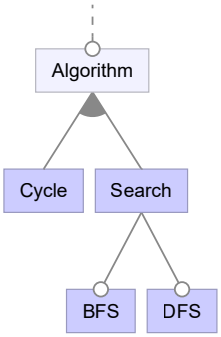
(b) *createFeatureBelow*(*BFS*, *GraphLibrary*)
createFeatureBelow(*DFS*, *GraphLibrary*)



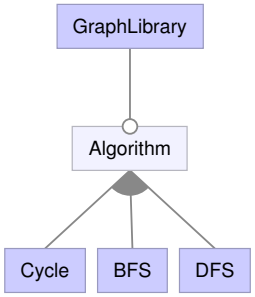
(c) *createFeatureAbove*(*Search*, {*BFS*, *DFS*})



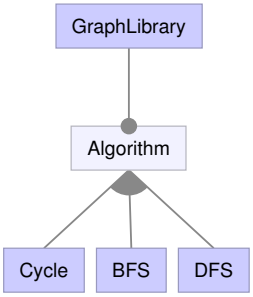
(d) *removeFeatureSubtree*(*EdgeType*)



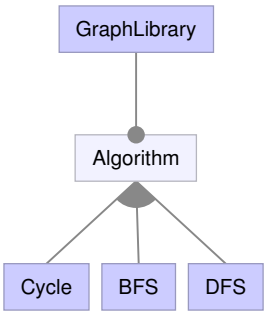
(e) *moveFeatureSubtree*(*Search*, *Algorithm*)



(f) *removeFeature*(*Search*)



(g) *setFeatureOptional*(*Algorithm*, *false*)



$BFS \Leftrightarrow \neg DFS$

(h) *createConstraint*(C_1, $BFS \Leftrightarrow \neg DFS$)

Figure 3.3: Applying compound operations subsequently on a feature model. The *FM* parameter has been omitted for brevity, and we label features with their IDs.

In Figure 3.3, we show an integrated example of applying compound operations to a feature model. Starting with the initial feature model in Figure 3.3a, compound operations are subsequently applied in Figure 3.3b–3.3h.

As mentioned above, additional compound operations may be designed to allow for new functionality. Depending on the introduced operation, this may require that the primitive operations or the conflict detection be adjusted.

3.2.6 Operation Application

Firstly, primitive and compound operations are only plain metadata that describe changes to feature models. This is useful for exchanging operations among sites, but does not actually specify how operations are applied to feature models to produce new, changed feature models as shown in Figure 3.3. For that, we define an application function for compound operations.

Definition 3.5. *Let $FM \in \mathcal{FM}$ be a legal feature model. Further, let PO and CO be a primitive and a compound operation whose preconditions are satisfied with regard to FM. In addition, $FM' = applyPO(FM, PO)$ denotes the feature model FM' that results from applying PO to FM. We then define $applyCO(FM, CO)$ as the subsequent application of all primitive operations contained in CO to FM with $applyPO$.*

We assume *applyPO* to be provided by the implementation; it should ensure all postconditions of primitive operations. Because *applyCO* only relies on the implementation-specific *applyPO*, none of the compound operations defined above has to be treated specially.

Further, we can express *multi-target operations* (operations that affect a selection of features or cross-tree constraints) as compound operations. A multi-target operation can be broken down into many single-target operations. For example, a multi-target operation that removes three features *Directed*, *Undirected* and *Cycle* can be expressed as three subsequent single-target operations that remove the respective feature subtrees. In other words, two compound operations CO_a and CO_b can be composed by concatenating their primitive operation sequences, yielding a new compound operation which we denote as $composeCOs(CO_a, CO_b)$. To compose CO_a and CO_b, it is required that their preconditions are satisfied with regard to FM and $FM' = applyCO(FM, CO_a)$, respectively. Thus, the composed CO's preconditions are also satisfied with regard to FM. By subsequently using *composeCOs*, arbitrary multi-target operations can be expressed, as long as no composed operation violates preconditions of a subsequently composed operation (which can be ensured with a suitable user interface). Composed operations can then be applied with *applyCO* just like simple compound operations. For example, the deletion of three selected features can be expressed as a single compound operation with *composeCOs*:

$$\begin{aligned} &composeCOs(composeCOs(removeFeature(FM, Directed),\\ &\qquad\qquad removeFeature(FM, Undirected)),\\ &\quad removeFeature(FM, Cycle)) \end{aligned}$$

As the removal operations do not affect each other, the preconditions are properly satisfied and the operations may even be generated on the same feature model FM.

From the definitions of the application function *applyCO* and the compound operations, we can derive a first theorem that characterizes the correctness of our system.

Theorem 3.1. *Let $FM \in \mathcal{FM}$ be a legal feature model. Further, let CO be a compound operation whose preconditions are satisfied with regard to FM. Then, $FM' = applyCO(FM, CO) \in \mathcal{FM}$, i.e., FM' is again a legal feature model.*

This theorem essentially states that our definition of feature models and operations always yields legal feature models in a single-user scenario: In that case, operations are applied immediately after generation and the operation history is linear. The theorem lays the groundwork for building a multi-user editor as well, as feature model legality is a basic consistency property which our editor must ensure. Therefore, it guides the design of the conflict detection described in Section 5.1. We sketch a proof for Theorem 3.1, which mostly relies on the preconditions of compound operations.

Proof. First, FM' still has *unique identifiers* for features and cross-tree constraints: Only compound operations that include *create* POs can affect $\mathcal{F}^{ID}_{FM'}$ and $\mathcal{C}^{ID}_{FM'}$. For these COs, preconditions such as $F^{ID} \notin \mathcal{F}^{ID}_{FM}$ 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 3.3): 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 3.3. 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 $FP^{ID} \in \mathcal{F}^{ID}_{FM}$. (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., $\mathcal{F}^{ID}_{FM} \subseteq \mathcal{F}^{ID}_{FM'}$, we only have to consider *createConstraint* and *setConstraint*, both of which ensure valid constraints as per precondition $Var(\phi) \subseteq \mathcal{F}^{ID}_{FM}$.

Next, the *single root* 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. □

3.3 Summary

In this chapter, we laid the groundwork for our concept by introducing a suitable feature model representation and operation model. Our feature model representation expresses features and cross-tree constraints as simple collections of attributes. Features and cross-tree constraints may then be created and updated with primitive operations, which we use to build higher-level compound operations suitable for the concurrency control scheme we introduce in Chapter 5.

4. Choosing a Concurrency Control Technique

In Section 2.2.2, we motivated the need for concurrency control in collaborative real-time editing systems to guarantee the CCI properties. Different concurrency control techniques have been proposed to achieve consistency maintenance. The goal of this chapter is to identify a concurrency control technique that is well-suited for a collaborative feature model editor. First, we specify more precisely what requirements a suitable concurrency control technique should satisfy. Then, we discuss several concurrency control techniques from the field of CSCW with regard to the identified requirements. Finally, we choose one that is appropriate for collaborative feature modeling and explain it in detail.

4.1 Requirements Analysis

First and foremost, the goal of editor software is to enable users to edit documents effectively and efficiently. Below, we identify several requirements for a collaborative feature modeling editor that serve this major goal. We take these requirements into account when choosing a concurrency control algorithm in the following sections.

Concurrency. A suitable collaborative feature model editor should be able to accommodate multiple collaborators that concurrently access and edit a feature model [DCS94, EGR91, GM94]. In particular, the system should not make collaborators reluctant to issue operations—for example, because they fear a possible negative outcome due to competing concurrent operations. Further, if the system does not take any precautions regarding concurrency control, this will inevitably lead to confusion and inconsistency so that users become wary towards the system, thus diminishing editing efficiency. Instead, a suitable system should be able to process operations with arbitrary causality and concurrency relationships (as we defined in Section 2.2.2).

Optimism. Regarding the level of optimism discussed in Section 2.2.1, we prefer optimistic over pessimistic approaches. This is because they provide a more responsive editing experience for collaborators and do not force collaborators to be

connected at all times, reducing user frustration [BBA98, DCS94, GM94, Pra99]. Optimistic approaches generally make one particular assumption about the system's mode of operation: Namely, that concurrent operations (operations generated simultaneously by different collaborators) are rarely in conflict, for example by modifying the same target object. Optimistic systems rely on this assumption to provide immediate feedback in the regular case where operations do not interact with each other in unexpected ways. However, if the probability for conflicting operations is high, optimistic systems have no advantage over pessimistic systems because conflict resolution requires tight synchronization.

In most editing domains (including feature modeling), the potential for conflict grows with larger groups of collaborators, smaller edited documents, and less coordination among collaborators. Experience has shown that in a small group of collaborators (which is the group size targeted by the CSCW research field), the conflict potential is low if all collaborators regularly synchronize with each other [Cam00, DDIZ18, EGR91, SFB+87]. This imposes an additional condition on our system, namely that collaborators may only be offline for short periods of time (e.g., in case of a network outage). We do not see this as a limitation, as collaborative *real-time* editors are intended for synchronous collaboration—regarding asynchronous modes of collaboration, much work has been done in the context of version control systems (cf. Section 2.3).

Intention Preservation. In modeling or specification activities, a crucial requirement for any editing software is that it accurately reflects the issued operations' intentions in the edited model (corresponding to the *intention preservation* property discussed in Section 2.2.2) [GM94, SC02, SJZ+98]. That is, when a collaborator submits an operation, she expects the system to apply and retain that operation's effect; especially after operations from any other collaborators have been processed. This requirement applies to, for example, CAD applications [BvdBB01, TCD07] and modeling of software artifacts with the Unified Modeling Language (UML) [LFST07]. In particular, models and specifications often have very clear semantics in their respective domain and it is important that they do not contain any inconsistencies or anomalies [Cam00]. For example, UML diagrams (such as class diagrams) may directly influence a software system by generating source code. In that case, if collaboration activities were to introduce unexpected changes (or reject already performed ones), this might have detrimental consequences for the generated software. Thus, a collaborative modeling editor must not ignore, reject or override any operations without notice (preferably, not at all).

Considering feature modeling, we argue that intention preservation is just as important as in other modeling activities. Because the feature model is a central artifact in a feature-oriented SPL, changes may impact the SPL in important ways: First and foremost, a feature model defines the set of valid products in an SPL. Thus, small changes (such as removing a feature) can impact the variability greatly (e.g., halve the number of products) and in unpredictable ways (due to arbitrary cross-tree constraints). Further, feature models are essential for communicating variability with stakeholders and developers and, thus, affect management tasks as well as the actual source code. Assuming that collaborators' own edits to the feature model are well-justified, we only have to consider the interaction of multiple collaborators,

which do not necessarily coordinate their actions. Consequently, our system must ensure that operations issued by multiple collaborators simultaneously do not affect the edited feature model in unexpected ways (such as rejected, masked or overridden operations).

Flexibility. As mentioned in Section 2.1.1, there are several types of feature models, which allow representing different kinds of variability. Although we focus on basic feature diagrams in this thesis, a concurrency control technique also suited for extensions is desirable. In particular, the feature model representation and operation model (discussed in Section 3.1 and 3.2, respectively) should allow for later extension (e.g., cardinality-based feature models or additional operations). Further, as collaborators may have different perceptions of *conflict*, the rules for conflict detection are not obvious. In particular, the SPL research community has identified various consistency properties and anomaly analyses on feature models, such as dead feature analysis (cf. Section 2.1.2). A suitable concurrency control technique should be able to leverage these existing analyses so that collaborators can adjust the conflict detection algorithm to their particular needs.

Correctness. Past research has shown that building optimistic systems which satisfy the CCI properties (introduced in Section 2.2.2) is a challenging task [IROM06, RBIQ13, SXA14]. In particular, showing the correctness of such systems requires significant effort, which, when omitted, leads to defective systems [IROM06]. We therefore prefer concurrency control techniques that have been proven to comply with the CCI model and can be considered mature because they have been applied repeatedly (and successfully) in the literature. Thus, we have to reason only about the correctness of our additions specific to feature modeling, which facilitates our system design.

4.2 Concurrency Control Techniques

In this section, we discuss common concurrency control techniques with regard to the requirements we identified for collaborative feature modeling. For each technique, we briefly describe the idea behind it and its strengths and weaknesses, focusing on the feature modeling domain. First, we discuss concurrency control techniques that prevent conflicting operations (i.e., operations that violate each other's intentions, for example by writing to the same part of a feature model). Conflict-preventing techniques attempt to circumvent conflicts by constraining when and by whom operations may be generated. Further, we discuss other influential techniques that attempt to process conflicts when they emerge. We are not aware of any comprehensive survey of concurrency control techniques for real-time collaborative editing; thus, we include techniques commonly mentioned in the literature.

Turn-Taking. A simple technique to avoid conflicts altogether is to ensure that only one collaborator can write to the edited feature model at any point in time. This technique, known as *turn-taking* (or *floor control*), assigns only one collaborator with a *token*, which designates that she may edit the feature model [EGR91, Gre91]. If another collaborator wants to submit an edit, he may ask the editing collaborator to pass him the token, which makes him the new editing collaborator. This technique

can be applied to any single-user editor with minimal effort to provide some basic collaboration facilities (e.g., with remote desktop software).

Although simple to implement, this technique does not allow any concurrent editing in the system at all. Worse, the editing collaborator may hold on to the token for any period of time, in which case no other collaborator can submit any edit. (This can be solved in a client/server architecture by providing the server with special authority regarding the token, which can be considered a variant of the locking technique discussed below.) In addition, when screen sharing software is employed, all users are forced to use the same view, which unnecessarily limits users that prefer different feature model visualizations. This technique also requires additional communication effort to pass, request, and release the token, diminishing optimism in the system. Overall, turn-taking is too limiting for collaborative feature modeling in terms of concurrency and optimism.

Locking. Another popular technique involves *locking* an object before an operation concerning it may be generated [EN10, GM94, MO92]. When exclusive locks are employed, a collaborator may only submit a change if she previously obtained a lock on said object. Thus, no other collaborators may simultaneously write to the locked object. The described approach is the most basic locking approach, referred to as *pessimistic locking*, because a lock request always has to be sent (and approved) before editing an object. In addition, several *optimistic locking* approaches have been proposed in the literature. They all have in common that they do not guarantee the CCI properties on their own, but extend another concurrency control technique to provide additional benefits. This includes *optional locking*, which extends the operational transformation approach (discussed below) to additionally ensure data integrity [CS01a, Sun02]; *multiple granularity locking*, which introduces *intention locks* to propagate locks to objects of coarser granularity [MD96, Pap02]; and *post-locking*, which has been devised to resolve conflicts *after* they have occurred, contrary to traditional *pre-locking* approaches [XO05].

With regard to feature modeling, pessimistic and optimistic locking approaches suffer equally from the question of *lock granularity* [DCS94]. As mentioned, locks are placed on *objects* in the edited document. *Object* may refer to an individual feature, an entire feature subtree or the entire feature model (in- or excluding the concerned cross-tree constraints). Clearly, locking the entire feature model does not permit any concurrency at all. Locking individual features, on the other hand, seems more promising, as this has minimum impact on simultaneous edits. However, it is not clear which features have to be locked for each feature modeling operation the system should support. For example, when moving feature subtrees in the feature model, special care has to be taken so that no cycles are introduced to the feature tree (discussed in detail in Section 3.2), which would require that entire feature subtrees are locked. In other words, a locking model has to be designed carefully for each operation that is introduced to the system, limiting flexibility.

Further, locking cannot guarantee any non-syntactic consistency properties of feature models (such as dead features and other analyses discussed in Section 2.1.2). This is because the conflict potential has to be analyzed *when designing the system* to be able to prevent conflicts. Semantic consistency properties, however, depend

on complex feature relationships formed by the feature tree and cross-tree constraints, which cannot be predicted at design time. For example, the most basic semantic consistency property involves checking whether a feature model yields any valid products at all (*void feature model* analysis). This usually involves invoking a reasoning engine (e.g., a SAT solver [MWC09]) on a feature model including the potentially problematic operation's effect. However, to determine which parts of the model should be locked, locking cannot possibly predict the outcome of the reasoning engine before the operation has even been generated. Thus, locking alone can only guarantee basic syntactic consistency of a feature model and no properties of the modeled SPL.

Although locking can guarantee intention preservation in the sense that concurrent operations will never interact in unexpected ways, it does so by rejecting any operations that target already locked objects. As stated in the requirements, this is not satisfactory for users because their operations may be rejected—and from their perspective, without good reason, as lock acquisition can appear arbitrary to users. In summary, locking is not a very compelling approach for feature modeling because it makes compromises regarding the optimism, intention preservation, and flexibility requirements.

Conflict-Free Replicated Data Types (CRDTs) seek to simplify the reconciliation process in distributed systems (including collaborative real-time editors) to achieve eventual consistency [SPBZ11a, SPBZ11b]. CRDTs use simple mathematical properties to guarantee that all replicas converge. *State-* and *operation-based* CRDTs can be distinguished, which correspond to the distinction introduced in Section 2.2.1. That is, a state-based CRDT always transmits its entire state to other sites, while an operation-based CRDT only transmits operations that describe a change.

CRDTs are advantageous over other techniques in that they are lightweight and simple to implement, without requiring any pessimistic synchronization between sites. However, as CRDTs require strong mathematical properties, designing CRDTs for more complex applications is difficult [SPBZ11a]. For example, operation-based CRDTs are not suitable for feature modeling because they require that operations commute, which is not generally the case in a feature model editor (cf. Section 3.2). Further, both operation- and state-based CRDTs basically assume that no conflicts occur (thus, *conflict-free* replicated data types) or that the problem statement can be recast without conflicts. For example, the value of the CRDT *last-writer-wins register* is always determined from the most recent write—thus, conflicting operations just override each other. However, as the intention preservation requirement states, we prefer not to override operations without notice.

Overall, CRDTs are primarily suitable for replicating small, simple data structures for which conflict potential is low or can be ignored. However, CRDTs do not provide a concept for accommodating conflicts while preserving user intentions, as required by collaborative feature modeling.

Serialization. is a basic approach to enforce convergence in operation-based collaborative editors [BG93, GM94, LK04]. In contrast to CRDTs, serialization can also guarantee convergence for operations that do not commute. The idea is simple:

Operations can be generated concurrently, but are timestamped in such a way that a total order is formed (e.g., using Lamport timestamps [Lam78]). Operations will then be executed at all sites according to this particular total order, thus ensuring convergence. In pessimistic systems, this can be implemented by delaying the execution of an operation until all preceding operations have arrived. Alternatively, operations can also be applied optimistically, in which case an arriving operation may necessitate that operations are "reordered" (i.e., undone and redone to ensure the total order). This way, serialization can guarantee the convergence property.

Unfortunately, this technique suffers from the same problem as CRDTs regarding the intention preservation requirement: Conflicting operations are neither prevented nor detected. Instead, they are simply applied according to the total order, therefore they might have unexpected influence on the edited feature model. For example, suppose two sites A and B in an optimistic system generate and apply operations O_1 and O_2, respectively, which both set the same feature's attribute. Assuming that O_1 precedes O_2 according to the total order, site A will simply apply O_2 when it arrives, thus overriding O_1's effect. Site B, however, has to reorder the operations by undoing O_2, then applying O_1 and O_2 so that the total order is obeyed. From the perspective of the user at site A, this will look like her own operation has been successfully applied first, only to then be undone or rejected, causing confusion in the process. Further, the total order imposed on operations is usually arbitrary and not clear to the users. Thus, the serialization technique is not suitable for feature modeling.

Operational Transformation (OT). From the necessity to accommodate non-commutative operations, the OT technique emerged [SE98, SJZ+98]. OT is a mathematical framework for optimistic operation-based editors which guarantees convergence and intention preservation for non-conflicting operations. Historically, research on collaborative real-time editors has focused on the text editing domain [EG89, SJZ+98, SXSC04]. In this domain, it is customary to represent a text document as a position-indexed sequence of characters. Operations such as *inserting* or *deleting* a character then refer to a specific position. For example, $insert(6, ",")$ will insert a comma character at the sixth position of a document (counting from 1). This is a natural representation of a text document. However, concurrent operations complicate matters because optimistic systems may execute them in different orders.

We illustrate the problem with the example shown in Figure 4.1. Suppose that two collaborators A and B are editing a document containing the text `Hello World`. Both collaborators simultaneously generate operations: User A generates and applies $insert(6, ",")$, arriving at `Hello, World`. Meanwhile user B generates and applies $insert(12, "!")$ and arrives at `Hello World!`. Both transmit their operation to the other user via network. If both operations are simply applied unchanged, A and B will arrive at `Hello, Worl!d` and `Hello, World!`, respectively. Two observations can be made here: First, both sites arrived at different documents, thus violating the convergence property. Second, at site A the intention of B's operation has been violated, because B intended to insert an exclamation mark *after* the word `World`, not inside it. This happens because the operations' *generation* context does

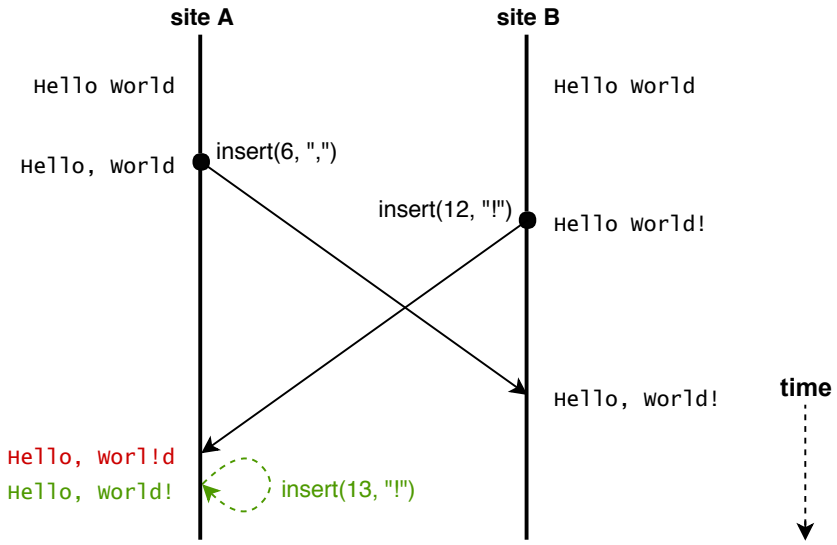


Figure 4.1: Operational transformation in a collaborative text editing scenario. The sites' documents at a time are shown to the left and right. Red and green colors refer to the results achieved without and with OT, respectively.

not match their *execution* context at the other site, because the locally generated operation invalidates the arriving operation's position parameter.

To solve this problem, OT can be used to change an operation's context to match its execution context. In this example, we may transform arriving operations against locally generated operations by adjusting the position parameter accordingly. To do so, we define a *transformation function* for transforming *insert* operations against other *insert* operations: Whenever an arriving operation targets a position *after* a position targeted by one or multiple locally generated operations, its position is incremented by the number of locally generated operations. In this example, one operation has been generated locally at each site. At site A, $insert(12, "!")$ arrives. This operation targets the position 12, which is after 6, the position targeted by the locally generated $insert(6, ",")$. Therefore, A increments the targeted position and executes $insert("!", 13)$, arriving at `Hello, World!`. At site B, $insert(6, ",")$ arrives. This operation targets the position 6, which is not after 12, thus B may simply apply the operation unchanged and arrives at `Hello, World!` as well. Using the OT technique, both sites converged, and all intentions were preserved.

Operational transformation is well-researched and used in a number of research prototypes [EG89, NCDL95, RNRG96, SXSC04]. Although the basic idea of OT is quite intuitive, designing and implementing an OT system poses some difficulties. For each pair of operations in the systems, a transformation function has to be designed and verified. Thus, the number of transformation functions grows quadratically with the number of operations in the system. Further, in the text editing domain, character-wise insert and delete operations provide only the most basic functionality required by users. Thus, extending OT systems with updates (e.g., to allow formatted text) or string-wise insert and delete operations (e.g., to remove a selected range of characters) is useful, but not trivial [SXSC04]. This obstructs our goal to build a rich feature modeling editor with potential for future extension.

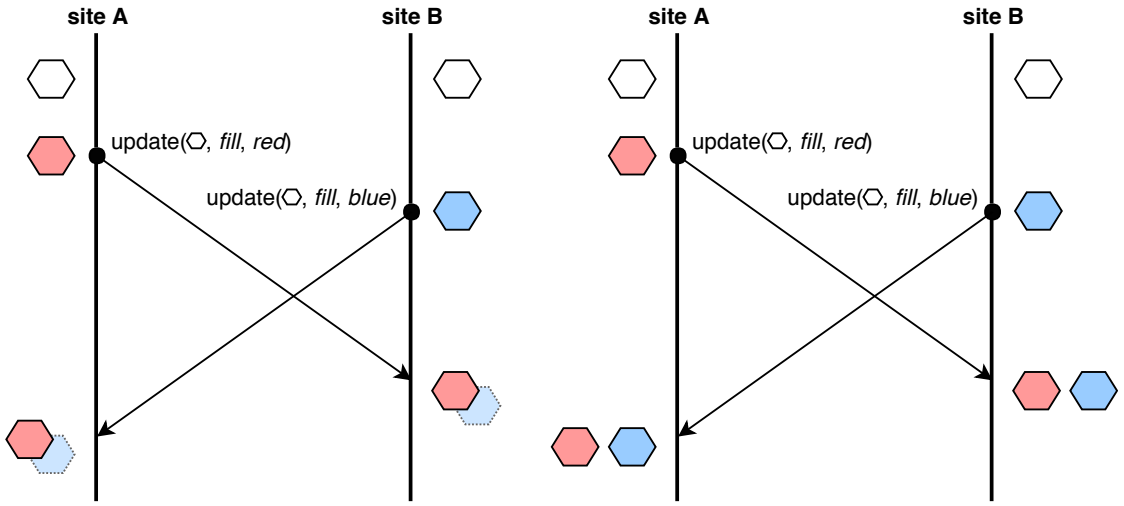
Unsurprisingly, this complexity also hampers reasoning about the correctness of OT systems. Verification of transformation functions and control algorithms is necessary, as many subtle problems are hard to locate otherwise [SXA14]. For example, the text editing systems which pioneered OT in a peer-to-peer [EG89] and client/server architecture [NCDL95] both do not satisfy the CCI properties [Cor95b, SS99]. Using OT for a feature model editor therefore not only requires design, but also verification of transformation functions and their interaction with transformation control algorithms.

Further, most OT algorithms are only applicable in domains where the edited document can be decomposed into a linear (i.e., one-dimensional) address space, such as text documents. Feature diagrams, however, use a tree structure that cannot be linearized in a natural way. Extensions for higher-dimensional address spaces have been proposed, but cannot profit from traditional OT techniques.

Just like the other techniques discussed above, OT cannot handle conflicting operations well [Che01]. Suppose, for example, one collaborator inserts a word inside a paragraph of text. Meanwhile, another collaborator simultaneously removes said paragraph. We consider this situation a conflict, as both operations cannot be applied in the same time without violating the intention of the other. In OT systems, such conflicts are usually resolved automatically with an arbitrary policy, such as preferring delete operations over insert operations at all times, or preferring operations from users with a higher priority. Thus, OT can guarantee intention preservation only in the narrow sense that concurrent operations will not interact in unexpected ways. However, the user intention cannot always be preserved, as operations may be effectively ignored or overridden by others. We conclude that operational transformation is a powerful, but complex approach that cannot guarantee our requirement of intention preservation.

4.3 Multi-Versioning Techniques

In the discussion above, a recurring issue is that techniques are lacking when it comes to preserving intentions of conflicting operations. The underlying problem is that all of the discussed techniques either reject conflicting operations (turn-taking, locking) or resolve conflicts automatically using an arbitrary policy such as *last writer wins* (CRDTs, serialization, OT). Instead, we argue that collaborators should be *involved in the conflict resolution process*, thus requiring a *manual conflict resolution approach*. By adopting a manual approach to conflict resolution, collaborators have fine-grained control over the outcome of a conflict situation [Che01, SFB+87, Wul95]. We have already discussed the importance of intention preservation with regard to feature modeling and how small changes can have a big impact on the modeled SPL. Manual conflict resolution can help to investigate conflict situations and raise the collaborators' confidence in the resulting feature model, because the resolution process is completely transparent to the collaborators. Of course, collaborators do not want to spend all their time resolving conflicts. Thus, a suitable conflict detection mechanism has to be designed to determine which operations are conflicting at all. Further, the conflict resolution process, although manual, must be clear, intuitive, and easy to use.



(a) Multi-Version Single-Display (b) Multi-Version Multi-Display

Figure 4.2: Multi-versioning techniques in a collaborative graphics editor. The sites' documents at a time are drawn as shapes to the left and right.

To allow for such a manual conflict resolution process, we use *multi-versioning*, which shares some similarities with version control systems. Multi-versioning techniques, in contrast to the single-versioning techniques discussed above, accommodate conflicting operations by creating multiple versions of the concerned object. Collaborators are then given the opportunity to choose a version they want to continue to work with. Just as their single-versioning counterparts, multi-versioning techniques have to be designed carefully because in an optimistic system, every collaborator may submit operations at any time. Sun et al. [SXSC04] distinguish two kinds of multi-versioning techniques, *multi-version single-display* and *multi-version multi-display*, which we illustrate in Figure 4.2.

Multi-Version Single-Display (MVSD) techniques manage multiple versions of an object internally, but expose only one version in the user interface. This concept was originally introduced by Sun et al. [SXSC04] in the context of collaborative word processing.

In their *CoWord* approach, Sun et al. extend the operational transformation approach discussed above with *update* operations. They did this to make *Microsoft Word* collaboration-aware. However, they discovered that concurrent update operations on the same object (e.g., a word or an image) may conflict. They then devised the MVSD technique to allow manual resolution of conflicting operations. For example, suppose two collaborators simultaneously change the fill color of a shape to red and blue, respectively (shown in Figure 4.2a). In contrast to the traditional OT technique, the MVSD approach creates two versions of the shape, one with red and one with blue fill color. Because the architecture of the underlying Microsoft Word application does not allow for multiple object versions to be displayed simultaneously, only one of these versions is displayed initially (which one, red or blue, is subject to an arbitrary policy). The other object version is initially masked, but can be recovered with the OT-specific *AnyUndo* technique [Sun03].

We have already discussed the operational transformation approach that MVSD extends with regard to feature modeling. Regarding the requirement of intention preservation, the MVSD approach is a step in the right direction, as it allows conflicting operations to be preserved in multiple versions. However, the user experience is still poor, as one version is arbitrarily favored over the others, and switching versions involves using undo. More importantly, the MVSD approach is explicitly designed for update conflicts which occur when two update operations target the same object's attribute. Feature modeling, however, requires a broader definition of conflict. For example, feature modeling operations with different targets can indeed conflict (e.g., when a feature is removed, but a cross-tree constraint referring to that feature is simultaneously added). We discuss this in detail when designing the conflict detection in Section 5.1. Overall, we conclude that multi-version single-display approaches are only partially suitable for feature modeling, because we are not aware of a MVSD technique that allows for a flexible conflict detection.

Multi-Version Multi-Display (MVMD). The other kind of multi-versioning technique, Multi-Version Multi-Display, also creates multiple versions of objects when faced with conflicting operations [Che01, SC02]. In contrast to MVSD, however, MVMD techniques display all created object versions to the user. Thus, no version is favored and the responsibility of choosing a final object version lies completely with the collaborators. The MVSD approach has been pioneered in the *Tivoli* system, which is a whiteboard meeting-support tool [MMvM+95]. In Tivoli, multiple versions are created whenever two concurrent operations target the same object. Sun and Chen [SC02] have significantly expanded upon this approach, including a formalization, a multi-versioning algorithm, and an implementation of the Graphics Collaborative Editing (GRACE) system. Their approach includes a more precise conflict detection and can handle any number of collaborators, in contrast to Tivoli, which only allows two collaborators.

In GRACE, a document consists of multiple *objects* (e.g., lines or circles). These objects have attributes (such as stroke width or fill color) which may be changed by operations. In contrast to OT, objects are targeted using designated identifiers rather than positional indices. An update operation then consists of the targeted object's identifier, the updated attribute and the new attribute's value. As long as concurrent operations target different objects and attributes, GRACE can simply apply arriving operations to the document. For example, consider a circle and a rectangle, edited by two users A and B. Both collaborators simultaneously generate operations: User A fills the circle with red, and user B fills the rectangle with blue. Both transmit their operation to the other user. The other site's operation can then simply be applied: That is, user A first fills the circle with red, then receives B's operation and fills the rectangle with blue. Meanwhile, user B first fills the rectangle with blue, and after receiving A's operations fills the circle with red. Both sites have converged and preserved all user intentions without need for reordering or transformation. This is because GRACE's document model enforces that non-conflicting concurrent operations commute and can be applied in any order.

However, there is one case when operations do not commute: Namely, whenever concurrent operations target the same attribute on the same object. In that case, the last applied operation would override all previous operations' effects. To avoid this

	Turn-Taking	Locking	CRDTs	Serialization	OT	MVSD	MVMD
Concurrency	○	◐	●	●	●	●	●
Optimism	◐	○	●	●	●	●	●
Intention Preservation	●	○	○	○	○	◐	●
Flexibility	●	○	○	○	○	○	◐
Correctness	●	◐	◐	●	○	○	◐

Table 4.1: Comparison of concurrency control techniques.

effect and preserve all intentions, GRACE considers these operations conflicting and therefore creates multiple versions of the targeted object, to which it then applies the operations. Reusing the example given for MVSD above, suppose two collaborators simultaneously change the same shape's fill color to red and blue, respectively (shown in Figure 4.2b). When receiving the other collaborator's operation, GRACE detects that the operations are in conflict, therefore creating two versions of the shape with red and blue fill colors, respectively. The users can then proceed with editing as usual, keeping both objects or choosing one by deleting the other. By creating multiple versions, GRACE ensures that concurrent operations on the same object always commute. Thus, reordering or transformation approaches as discussed above are unnecessary in GRACE.

In Table 4.1, we show all discussed concurrency control techniques and how they relate to our requirements defined in Section 4.1. We use ○, ◐ and ● to denote barely, partially and completely fulfilled requirements. For correctness, we depict whether much work is required (○) or whether correctness is trivial to show (●). We have found that the MVMD approach taken in GRACE suits our requirements and is, with some adjustments, applicable to feature models:

- **Concurrency**. The multi-versioning algorithm employed by GRACE (detailed in the next section) can handle any number of collaborators, which may submit operations with arbitrary causal relationships.
- **Optimism**. The algorithm is designed specifically with optimism in mind, by building on the fact that non-conflicting operations can be applied in any order.
- **Intention Preservation**. Competing operations from different collaborators are either not in conflict and can simply be applied, or result in multiple versions, which can then be manually inspected by the collaborators to choose the best outcome. No operations are ever rejected, masked or overridden.
- **Flexibility**. By making a few adjustments to the way the multi-versioning algorithm is applied, the approach becomes extensible, allowing other feature modeling representations and operations.

- **Correctness**. For our intended usage, the multi-versioning algorithm has been proven to be correct. We further reason about the correctness of our adjustments in Section 5.4.

Of all the techniques we investigated in this chapter, the MVMD approach taken in GRACE satisfies our requirements best. We proceed to describe the multi-versioning algorithm used in GRACE in more detail. In the next chapter, we explain how this algorithm may be adjusted to make it suitable for collaborative feature modeling.

4.4 The MOVIC Algorithm

The Multi-Version Multi-Display technique as described in this section has originally been used in the GRACE system to maintain multiple versions of edited objects in the face of conflict. To create and maintain multiple object versions in an optimistic system, Sun and Chen devised a multi-versioning algorithm which groups operations according to whether they are *conflicting* or *compatible* [Che01, SC00, SC02]. At the core of the GRACE editor, the Multiple Object Versions Incremental Creation (MOVIC) algorithm is responsible for detecting conflicts between operations, guaranteeing convergence and intention preservation (cf. Section 2.2.2) and determining and creating appropriate object versions. We discuss some concepts fundamental to understanding the MOVIC algorithm, starting with the *conflict relation*.

Conflict Relation

The notion of *conflict* is central to any group editing software which seeks to guarantee the CCI properties. We have already given the following informal definition for conflicting operations in Section 2.2.2: *Two concurrent operations are* in conflict *if they have irreconcilable intentions or if together they influence the feature model in unexpected ways.* To be able to capture conflicts algorithmically, Sun and Chen introduce a *conflict relation* used by the MOVIC algorithm that specifies more precisely when operations are in conflict.

Definition 4.1 (GRACE Conflict Relation [SC02])**.** *Two operations O_a and O_b* conflict *with each other, denoted as $O_a \otimes O_b$, if and only if all of the following conditions are true:*

- *$O_a \parallel O_b$ (O_a and O_b are concurrent),*
- *O_a and O_b target the same object (e.g., a shape or image),*
- *O_a and O_b target the same attribute (e.g., fill color or border width), and*
- *O_a and O_b set that attribute to different values.*

If two operations O_a and O_b are not in conflict with each other, i.e., $O_a \oslash O_b$, they are compatible*, denoted as $O_a \odot O_b$.*

The GRACE conflict relation states that two concurrent operations are in conflict whenever they set the same object's attribute to different values. Note that, according to this relation, *create* operations can never cause conflicts because no other

operation can target an object that has only just been created. Further, only concurrent operations are considered conflicting. Thus, updating an object's attribute *after* another operation has updated the same attribute is perfectly valid, as expected by the users (because such operations are causally related, not concurrent).

Unfortunately, the GRACE conflict relation is not suitable for collaborative feature modeling: Suppose we applied this conflict relation in our feature model editor. We have already defined a representation for feature models akin to GRACE's document model in Section 3.1, where features and cross-tree constraints qualify as objects and *parentID*, *optional* etc. would be object attributes. The conflict relation may then be applied to primitive operations, in principle. However, the GRACE conflict relation only captures one kind of conflict (involving two concurrent updates of the same object's attribute). Although this is one important kind of conflict that may occur in collaborative feature modeling, conflicts also arise in other situations: Take, for example, the *moveFeatureSubtree* operation which requires that the move target does not descend from the move source, to guarantee that the feature model stays free of cycles. Suppose two collaborators concurrently move the features `A` below `B` and `B` below `A`, respectively. By the definition of *moveFeatureSubtree* in Section 3.2, two compound operations will be generated, each comprising an *updateFeaturePO* setting `A`'s *parentID* to `B` and `B`'s *parentID* to `A`, respectively. However, because both operations target different features, they will not be considered conflicting by the GRACE conflict relation—although they introduce a cycle to the feature model.

This example and other kinds of conflicts (expanded upon in Section 5.1) are not covered by this conflict relation. However, as has been noted by Xue et al. [XOZ03], the GRACE conflict relation is not the only possible conflict relation. In fact, any binary relation on operations can be employed as a conflict relation, provided it is irreflexive (no operation is in conflict with itself) and symmetric (from $O_a \otimes O_b$ follows that $O_b \otimes O_a$). Xue et al. leverage this fact to combine the MVMD approach with a *locking* technique; Sun and Chen too refine their approach with additional conflict relations [SC02]. Similarly, we introduce two conflict relations tailored to feature modeling in Section 5.1, which allow us to detect all situations in which a feature model's consistency may be compromised.

Combined Effect

In Section 4.1, we stated that a collaborative feature modeling editor should accommodate concurrent operations with arbitrary causality relationships and preserve all operations' intentions. To this end, Sun and Chen introduce several rules to specify the *combined effect* of a group of operations [SC02].

Suppose that a group of collaborators submits a group of arbitrary operations concurrently. Using a suitable conflict relation, we can determine which operations are in conflict by pairwise comparison. In the case that all operations are mutually compatible, they may all simply be applied to the same original object. Accordingly, if all operations are mutually conflicting, the only way to preserve all intentions is to create as many versions of an object as there are operations and apply each operation to its corresponding version. However, a group of concurrent operations may also contain arbitrary conflict relationships other than mutually compatible and conflicting, which complicates matters.

For example, consider three operations O_1, O_2, and O_3: One collaborator may set a feature's name to `A` (O_1). Simultaneously, another collaborator may set the same feature's name to `B` (O_2), but changes her mind and subsequently sets it to `C` (O_3), before O_1 has arrived. Thus, this group of operations has the conflict relationships $O_1 \otimes O_2$, $O_1 \otimes O_3$, and $O_2 \odot O_3$ (as O_2 and O_3 are not concurrent). In this situation, it is not clear how many versions the algorithm should create and on which versions operations have to be applied. Thus, Sun and Chen define the following three rules to determine the desirable combined effect of a group of operations:

- Given two operations O_a and O_b, if $O_a \otimes O_b$, they must be applied to two different versions of the targeted object. This rule ensures intention preservation.
- Given any two versions made from the same object, there must be at least one operation O_a applied to the first version and one operation O_b applied to the second version such that $O_a \otimes O_b$. This rule minimizes the number of versions, which facilitates the resolution process for the users.
- Given two operations O_a and O_b targeting the same object, if $O_a \odot O_b$, then their effects must be combined in at least one common version made from said object. This rule makes sure that an operation is applied to as many versions as possible without compromising consistency.

In summary, these rules guarantee that compatible operations are grouped in common versions, conflicting operations are applied to different versions and new versions are created only if at least one contained operation cannot be applied to all other versions. Based on these combined effect rules, Sun and Chen define concrete data structures which can be shown to achieve the specified combined effect.

Definition 4.2 (Combined Effect [SC02])**.** *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 $CG \subseteq GO$ of mutually compatible operations. A* compatible group set *$CGS = \{CG_1, CG_2, \ldots, CG_m\}$ is a set of compatible groups where all of the following conditions are true:*

- *for all operations $O \in GO$, $O \in CG_i$ for at least one $CG_i \in CGS$*
- *for all pairs of operations $O_a, O_b \in GO$, if $O_a \odot O_b$, $O_a, O_b \in CG_i$ for at least one $CG_i \in CGS$.*

A maximum compatible group *MCG is a compatible group where for all operations $O_a \in GO$ with $O_a \notin MCG$, $O_a \otimes O_b$ for at least one operation $O_b \in MCG$. A* maximum compatible group set *$MCGS$ is a compatible group set where all of the following conditions are true:*

- *every compatible group in $MCGS$ is a maximum compatible group*
- *all maximum compatible groups belonging to GO are included in $MCGS$.*

Given a maximum compatible group set $MCGS$, the combined effect *for GO is:*

- *for each $MCG \in MCGS$, one version of the targeted object is created*
- *given any $MCG \in MCGS$, all included operations are applied to the version corresponding to MCG.*

function MOVIC(O_i, $MCGS_{i-1}$)
  $MCGS_i \leftarrow \varnothing$
  $C \leftarrow |MCGS_{i-1}|$ ▷ counts CGs that are not fully conflicting with O_i
  while $MCGS_{i-1} \neq \varnothing$ **do** ▷ check O_i against every compatible group
    $CG_x \leftarrow$ any compatible group in $MCGS_{i-1}$
    $MCGS_{i-1} \leftarrow MCGS_{i-1} \setminus \{CG_x\}$
    if $O_i \odot CG_x$ **then** $CG_x \leftarrow CG_x \cup \{O_i\}$ ▷ step **A**
    else if $O_i \otimes CG_x$ **then** $C \leftarrow C - 1$ ▷ step **B**
    else ▷ step **C**
      $CG_{new} \leftarrow \{O \mid O \in CG_x \wedge O \odot O_i\} \cup \{O_i\}$
      $MCGS_i \leftarrow MCGS_i \cup \{CG_{new}\}$
    end if
    $MCGS_i \leftarrow MCGS_i \cup \{CG_x\}$
  end while
  if $C = 0$ **then** ▷ O_i conflicts with all CGs in $MCGS_{i-1}$, step **D**
    $CG_{new} \leftarrow \{O_i\}$
    $MCGS_i \leftarrow MCGS_i \cup \{CG_{new}\}$
  end if
  Let CG_{new} be any new CG created in steps **C** or **D**. ▷ step **E**
  For every such $CG_{new} \in MCGS_i$, if there is another $CG_y \in MCGS_i$,
  such that $CG_{new} \subseteq CG_y$, then $MCGS_i \leftarrow MCGS_i \setminus \{CG_{new}\}$.
  return $MCGS_i$
end function

Algorithm 1: MOVIC algorithm [SC02]. $O_i \odot CG_x$ and $O_i \otimes CG_x$ denote that O_i is compatible and conflicting with all operations in CG_x, respectively.

A Maximum Compatible Group Set essentially groups operations so that each group represents one version of the targeted object. As specified by the *combined effect*, this information can be directly used to create versions and apply operations appropriately.

Incremental Construction of an MCGS

It can be shown that the MCGS (and thus, the combined effect) for a given group of operations is unique [SC02]. This is essential for a collaborative real-time editor, as concurrent operations may be generated at any time, and the order of received operations may differ from site to site. To allow for unconstrained, optimistic collaboration, a distributed algorithm is required that incrementally constructs an MCGS. Because a group of operations' MCGS is unique, such an algorithm will always produce the same MCGS regardless of the arrival order, thus guaranteeing the convergence property.

To achieve this, Sun and Chen introduce the Multiple Object Versions Incremental Creation (MOVIC) algorithm (Algorithm 1). Whenever an operation O_i is generated or received, this algorithm constructs a new $MCGS_i$ from a previous $MCGS_{i-1}$ (where $MCGS_i$ is the Maximum Compatible Group Set corresponding to the first i processed operations). During the algorithm, O_i is added to existing MCGs and possibly new versions including O_i are created, both according to the combined

effect defined above. When a group of n operations has been received at all sites and the system is at quiescence, all sites will have converged to the same $MCGS_n$. This convergence property has been proven for groups of operations that target the same object [SC02].

Example 4.1. We give an example (replicated from Sun and Chen [SC02]) for applying the MOVIC algorithm on a group of four operations: O_1, O_2, O_3, and O_4, where $O_1 \otimes O_2$, $O_1 \otimes O_3$, $O_1 \odot O_4$, $O_2 \odot O_3$, $O_2 \odot O_4$, and $O_3 \odot O_4$.

We apply the MOVIC algorithm to these operations in two different arrival orders. For each operation, we list the corresponding $MCGS_i$ and the steps executed in Algorithm 1 (assuming that compatible groups in the MCGS are processed from left to right). First, suppose that the operations arrive in the order O_1, O_2, O_3, O_4:

$MCGS_1 = \{\{O_1\}\}$ ▷ step **D**
$MCGS_2 = \{\{O_1\}, \{O_2\}\}$ ▷ steps **B**, **D**
$MCGS_3 = \{\{O_1\}, \{O_2, O_3\}\}$ ▷ steps **B**, **A**
$MCGS_4 = \{\{O_1, O_4\}, \{O_2, O_3, O_4\}\}$ ▷ steps **A**, **A**

Two versions are created, one of which includes the effects of O_1 and O_4, while the other includes the effects of O_2, O_3 and O_4. Next, let operations arrive in the order O_1, O_2, O_4, O_3:

$MCGS_1 = \{\{O_1\}\}$ ▷ step **D**
$MCGS_2 = \{\{O_1\}, \{O_2\}\}$ ▷ steps **B**, **D**
$MCGS_3 = \{\{O_1, O_4\}, \{O_2, O_4\}\}$ ▷ steps **A**, **A**
$MCGS'_4 = \{\{O_1, O_4\}, \{O_4, O_3\}, \{O_2, O_4, O_3\}\}$ (before step **E**) ▷ steps **C**, **A**
$MCGS_4 = \{\{O_1, O_4\}, \{O_2, O_4, O_3\}\}$ ▷ step **E**

For $MCGS_4$, we have included the intermediate result $MCGS'_4$ to demonstrate how the compatible group $\{O_4, O_3\}$ (which has been created in step **C**) is absorbed by the superset $\{O_2, O_4, O_3\}$ in step **E** to guarantee a minimum number of versions. We observe that for both execution orders, the MOVIC algorithm arrived at the same result, which demonstrates its convergence property.

4.5 Summary

In this chapter, we presented a set of requirements we expect a collaborative real-time feature model editor to satisfy. We then discussed several concurrency control techniques with regards to these requirements, finding that many techniques cannot preserve all operation intentions. We introduced multi-versioning techniques to address this problem and identified a specific technique, Multi-Version Multi-Display, that satisfies our requirements. Finally, we described a concrete multi-versioning algorithm, which we adapt to collaborative feature modeling in the next chapter.

5. Concurrency Control for Collaborative Feature Modeling

In this chapter, we present our concurrency control approach for collaborative feature modeling. Our approach is based on the MVMD technique presented in the previous chapter, but includes several adjustments and additions. In particular, we discuss how the conflict detection provided by the MOVIC algorithm may be adapted to collaborative feature modeling. To this end, we utilize the formalization of feature models and operations we introduced in Chapter 3. Further, we extend the MOVIC algorithm with a voting technique that allows for fair and flexible conflict resolution. Moreover, we introduce a garbage collection scheme to make sure that our system is always responsive. Finally, we justify the correctness of our approach by showing that our system complies with the CCI consistency model.

5.1 Conflict Detection

In this section, we describe in detail how we adjust GRACE's MVMD approach to detect and accommodate conflicts in collaborative feature modeling editors. In particular, we motivate and describe the following strategies and mechanisms:

- **Global Targeted Object Strategy**. We manage a single global object to allow for intuitive conflict resolution while preserving all user intentions.
- **Causal Directed Acyclic Graph**. A data structure that captures causally-preceding relationships between operations for use in the conflict relations.
- **Outer Conflict Relation**. A conflict relation which guarantees that an appropriate feature model suitable for conflict detection can be produced.
- **Topological Sorting Strategy**. To produce such feature models, we use a topological sorting of operations according to their causal relationships.
- **Inner Conflict Relation**. A conflict relation that detects conflicting feature modeling operations using a set of conflict detection rules.

Finally, we demonstrate the interaction of these mechanisms with an integrated example.

5.1.1 Global Targeted Object Strategy

We have already seen that the MOVIC algorithm, as initially presented by Sun and Chen [SC02], distinguishes between *targeted objects* to determine whether operations are in conflict and to create versions of said objects which include only compatible operations. In a collaborative graphics editing system such as GRACE, an *object* might refer to a distinct shape, image, text or other graphical element on the screen.

In a collaborative feature modeling editor, objects may refer to individual features, entire feature subtrees, or the entire feature model (discussed briefly in the context of *lock granularity* in Section 4.2). Suppose each individual feature corresponds to an object managed by the MOVIC algorithm. In that case, conflicting operations will cause the targeted feature to be duplicated, so that the operations can be applied to their respective feature version. This may be fine in graphics editing systems; however, in the feature modeling domain, we have to consider the semantics of feature models: A feature model defines the set of valid products for an SPL, which needs to be carefully maintained and not changed by accident (according to our *intention preservation* requirement from Section 4.1). Automatically duplicating features whenever a conflict occurs would change a feature model's semantics by introducing new products (e.g., doubling the number of products). Thus, the conflict detection algorithm would actively interfere with the users' intentions and introduce unwanted products into the SPL. In addition, above we have already given an example where two *moveFeatureSubtree* operations on different features may produce a conflict. It is not clear how to resolve this kind of conflict by simply duplicating the involved features. Similar points can be made when considering entire feature subtrees as targeted objects.

Instead, we propose to only manage one *global targeted object* using the MOVIC algorithm, namely the entire feature model. This way, conflicts cause multiple versions of the feature model to be created, which can then be inspected by the users and discarded to determine the desired resolution outcome. Further, the algorithm does not touch any semantic properties of the feature model other than those intended by the users' operations. Using this approach, we can also detect and handle cases like the above-mentioned *moveFeatureSubtree* example (cf. Section 5.1.5). When discussing the *locking* approach in Section 4.2, we dismissed the global lock granularity because it forbids any concurrent operations in the system and essentially enforces a pessimistic approach. Under the assumption (made in Section 4.1) that operations are rarely in conflict, this is not a problem in our multi-versioning approach: As long as operations are compatible, all concurrency is automatically covered by one feature model version. Multiple versions are only created in the face of conflict. Even then, this approach is preferable to locking, as no operations are rejected by the multi-versioning approach, but rather arranged in multiple versions to facilitate later resolution.

In Section 2.3, we have already noted some similarities and differences of our approach compared to version control systems, making the point that version control systems are related to, but no solution to our specific needs. We further argue that the conflict management provided by the MOVIC algorithm is preferable to the three-way merge found in version control systems. This is because the MOVIC algorithm fits better with the ad-hoc nature of submitted operations in a collaborative

real-time editor (opposed to commits in a version control system): For a given set of operations with arbitrary conflict relations, the algorithm constructs a minimal set of versions, which accommodate these operations without requiring synchronization. Version control systems do not offer such facilities, merge conflicts have to be resolved one at a time. Further, the global targeted object approach taken here is simple and intuitive to understand; users do not need any background knowledge of version control systems.

5.1.2 Causal Directed Acyclic Graph

To construct an appropriate MCGS, the MOVIC algorithm requires that the causal relationships of operations are captured by the system. For this purpose, the *causally-preceding relation* adopted by Sun et al. [SJZ+98] may be used (described in Section 2.2.2). For two given operations, this relation states whether one *causally precedes* the other or if they are *concurrent*.

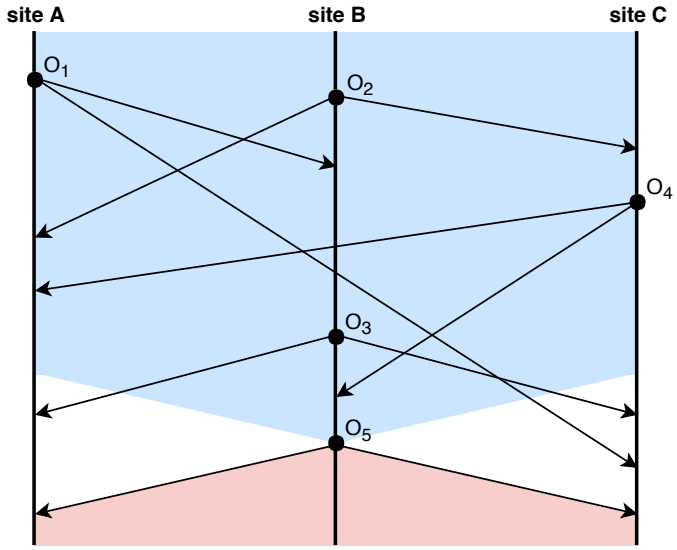
This relation suffices for conflict detection in GRACE, where it is utilized only in Definition 4.1. However, the *outer* and *inner conflict relations* for collaborative feature modeling (described in the following sections) need to retrieve further information about causality relationships. In particular, they need information about the causally preceding operations for a given particular operation.

To provide such functionality, we may utilize the fact that the causally-preceding relation is a strict partial order: The relation is irreflexive because no operation precedes (*happens before*) itself, and it is transitive by definition. Note that every strict partial order corresponds to a directed acyclic graph. Thus, we can construct a Causal Directed Acyclic Graph (CDAG) for any given group of operations [Mat88, SM94]. Because the causally-preceding relation is well-defined across all sites, the CDAG for a group of operations is unique. Based on the CDAG, we can specify the first set which the inner conflict relation relies on, namely the set of Causally Preceding (CP) operations for a given operation.

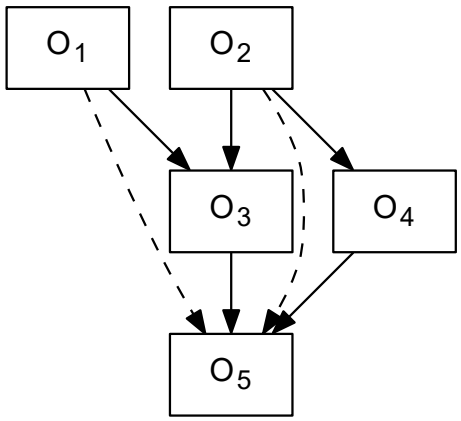
Definition 5.1. *Let GO be a group of operations. The* causal directed acyclic graph *for GO is the graph $G = (V, E)$ where $V = GO$ is the set of vertices and $E = \{(O_a, O_b) \mid O_a, O_b \in GO \wedge O_a \to O_b\}$ is the set of edges.*
Then, the set of causally preceding operations *for an $O \in GO$ is defined as* $CP_G(O) \coloneqq \{O_a \mid (O_a, O) \in E\}$.

Another piece of information required by the outer conflict relation is the set of Causally Immediately Preceding (CIP) operations for a given operation. An operation O_a causally *immediately* precedes another operation O_b when there is no operation O_x such that $O_a \to O_x \to O_b$. To obtain this set from a CDAG, we can use its *transitive reduction*, which removes all edges that only represent transitive dependencies [AGU72]. The transitive reduction of a CDAG is unique as well.

Definition 5.2. *Let GO be a group of operations and $G = (V, E)$ the causal directed acyclic graph for GO. 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* $CIP_G(O) \coloneqq \{O_a \mid (O_a, O) \in E'\}$.



(a) Group of Operations



(b) Causal Directed Acyclic Graph. Removing the dashed edges yields the transitive reduction.

Figure 5.1: Causal Directed Acyclic Graph for a group of operations.

We give an example for a CDAG and the associated CP and CIP sets.

Example 5.1. In Figure 5.1a, we depict three sites A, B, and C which generate and exchange a group of five operations with the following causal relationships: $O_1 \parallel O_2$, $O_1 \rightarrow O_3$, $O_1 \parallel O_4$, $O_1 \rightarrow O_5$, $O_2 \rightarrow O_3$, $O_2 \rightarrow O_4$, $O_2 \rightarrow O_5$, $O_3 \parallel O_4$, $O_3 \rightarrow O_5$, and $O_4 \rightarrow O_5$. For O_5, we depict its *cause* and *effect* in blue and red, respectively. The cause of O_5 corresponds to its CP set, which is fully known when O_5 is generated, whereas the effect of O_5 is yet unknown because it may expand in the future.

The matching CDAG is shown in Figure 5.1b. Every edge corresponds to one causally-preceding relationship. Thus, we can determine the operations' CP sets: $CP_G(O_1) = CP_G(O_2) = \varnothing$, $CP_G(O_3) = \{O_1, O_2\}$, $CP_G(O_4) = \{O_2\}$, $CP_G(O_5) = \{O_1, O_2, O_3, O_4\}$. By removing the dashed edges in the graph, we can obtain the CDAG's transitive reduction to determine the operations' CIP sets: $CIP_G(O_1) = CIP_G(O_2) = \varnothing$, $CIP_G(O_3) = \{O_1, O_2\}$, $CIP_G(O_4) = \{O_2\}$, $CIP_G(O_5) = \{O_3, O_4\}$.

Each collaborating site has a copy of the current CDAG which includes all previously generated and received operations. This CDAG is constructed incrementally by the system (similar to the MOVIC algorithm's MCGS): When an operation is generated or received, it is inserted into the CDAG according to its causal relationships with the other operations. The conflict relations for collaborative feature modeling described in the following sections can then make use of this CDAG's CP and CIP sets to detect conflicts.

5.1.3 Outer Conflict Relation

In the GRACE system, the conflict relation uses solely information provided by operations to determine conflicts: Two operations are in conflict when they are concurrent and target the same attribute on the same object, setting it to different values (cf. Definition 4.1).

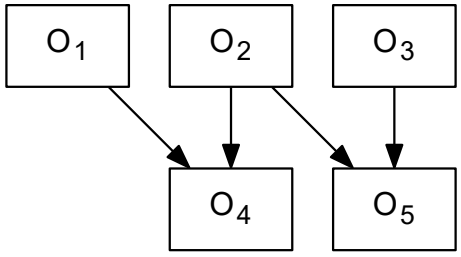


Figure 5.2: Scenario for determining an appropriate targeted feature model.

We have already discussed that this conflict relation is not suitable for collaborative feature modeling because it can only detect basic conflicts involving competing updates. Unfortunately, detecting other kinds of conflicts is difficult when the conflict relation may only inspect the involved operations' metadata. For example, to detect whether two *moveFeatureSubtree* operations are in conflict (i.e., introduce a cycle), a suitable conflict relation must consider the involved features' positions in the feature tree, which are not included in the operations' metadata. A possible solution to detect this and other kinds of conflicts is to include all metadata required for conflict detection in compound and primitive operations. This does not scale well with respect to the size of operations, which may end up including the entire feature tree; in addition, this approach would be highly operation-specific and guaranteeing arbitrary semantic properties of feature models would be difficult.

Instead, we propose that the conflict relation should not only consider operation metadata, but also the targeted feature model. Such a conflict relation could inspect the involved operations, find that they are concurrent *moveFeatureSubtree* operations and then proceed to apply them both to the targeted feature model, finally checking whether their application introduced a cycle—if so, the relation results in a conflict. This approach is possible due to our global targeted object strategy (cf. Section 5.1.1).

Designing such a conflict relation introduces a technical difficulty, namely finding the appropriate targeted feature model which two operations should be applied to. For example, consider a scenario with five operations depicted as a CDAG in Figure 5.2. Suppose the system checks whether the concurrent operations O_4 and O_5 are in conflict in the course of the MOVIC algorithm. Note that O_4 is generated in a context where O_1 and O_2 are in effect, whereas in the generation context of O_5, O_2 and O_3 are in effect. The question arises, which execution context is appropriate for determining a conflict between O_4 and O_5? The *intention preservation* property of the CCI model described in Section 2.2.2 states that an operation's execution context should match its generation context. Accordingly, to determine a conflict between O_4 and O_5, the feature model used by the conflict relation should include the effects of O_1, O_2, and O_3.

This, however, is only possible given two requirements: First, to apply O_4 and O_5 in this context, they must be compatible with O_1 O_2, and O_3. Because $O_1 \to O_4$, $O_2 \to O_4$, $O_2 \to O_5$, and $O_3 \to O_5$, these operations are considered compatible by a conflict relation. However, $O_1 \parallel O_5$ and $O_3 \parallel O_4$, therefore O_1 and O_3 may potentially conflict with O_5 and O_4, respectively. Thus, we must require that $O_1 \odot O_5$ and $O_3 \odot O_4$ before applying O_4 and O_5 in the context of O_1, O_2, and O_3. Second,

$O_1 \parallel O_3$, therefore O_1 and O_3 may also potentially conflict. If they do, they cannot be applied in the same context, in which case there is no meaningful context in which to apply O_4 and O_5 together. Thus, we additionally require that $O_1 \odot O_3$ before applying O_4 and O_5 in the context of O_1, O_2, and O_3.

This example shows that to check two operations for conflict, an appropriate targeted feature model must be prepared. However, this feature model is only meaningfully defined when all causally preceding operations of both 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. We specify how an appropriate conflict relation, termed $\otimes_O$, may satisfy this requirement.

Definition 5.3. *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:*

- *there is $O_x \in GO$ such that $O_x \to O_a$ and $O_x \otimes_O O_b$,*
- *there is $O_y \in GO$ such that $O_y \to O_b$ and $O_a \otimes_O O_y$, or*
- *there are $O_x, O_y \in GO$ such that $O_x \to O_a$, $O_y \to O_b$, and $O_x \otimes_O O_y$.*

This can be considered a kind of *transitivity property* which causes any conflict between two operations to be propagated to all causally succeeding operations. Compared to GRACE's original conflict relation $\otimes$, $\otimes_O$ ensures that there is a well-defined feature model for subsequent conflict detection, which enables us to check arbitrary consistency properties. However, $\otimes_O$ has the disadvantage that it may falsely flag operations as conflicting. For example, although O_4 causally succeeds O_1, it does not necessarily rely on the execution effect of O_1. Thus, O_4 and O_5 are flagged as conflict if $O_1 \otimes_O O_3$, although they may be perceived as compatible by collaborators. However, we expect such false positives to be rare because conflicts are assumed to be rare in general (cf. Section 4.1) and the height of the CDAG can be kept small with frequent synchronization and garbage collection (discussed in Section 5.3).

Note that $\otimes_O$ neither refers to any concepts specific to feature modeling, nor does it include a base case that specifies when operations are compatible. Instead, we distinguish between the *outer conflict relation* $\otimes_O$ and an *inner conflict relation* $\otimes_I$, discussed in Section 5.1.5. This way, we separate two concerns: $\otimes_O$ addresses the technical problem of ensuring an appropriate targeted feature model, while $\otimes_I$ contains conflict detection rules specific to collaborative feature modeling. Thus, $\otimes_O$ is also suitable for other domains than feature modeling which may require access to the targeted object.

To determine $\otimes_O$ for two given compound operations, we introduce outerConflicting (Algorithm 2), a recursive algorithm that satisfies Definition 5.3. With the Causally Immediately Preceding (CIP) sets of the CDAG defined in the previous section, it is straightforward to translate the transitivity property. In the base case, conflict detection is deferred to the inner conflict relation $\otimes_I$, which we discuss in Section 5.1.5. Because of the transitivity property, $\otimes_I$ is only used when $\otimes_O$ can guarantee that an appropriate feature model for conflict detection exists. We further define how to compute $\otimes_O$ using outerConflicting:

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

Algorithm 2: Computing the outer conflict relation $\otimes_O$ for compound operations.

Definition 5.4. *Two compound operations CO_a and CO_b are in* outer conflict*, i.e., $CO_a \otimes_O CO_b$, if and only if* OUTERCONFLICTING(G, CO_a, CO_b) = *true, where G is the current CDAG at the site that executes* OUTERCONFLICTING*. Otherwise, they are* outer compatible*, i.e., $CO_a \odot_O CO_b$.*

In Section 5.4, we show that $\otimes_O$ is indeed a conflict relation suitable for use within the MOVIC algorithm. Thus, we modify the MOVIC algorithm so that it uses $\otimes_O$ instead of the GRACE conflict relation $\otimes$. We move on to discussing our topological sorting strategy, which is needed to compute the inner conflict relation $\otimes_I$.

5.1.4 Topological Sorting Strategy

In the previous section, we discussed how the outer conflict relation $\otimes_O$ ensures the transitivity property so that an appropriate targeted feature model can be produced. We now explain the mechanism that produces these feature models, which is implemented by APPLYCOMPATIBLEGROUP (Algorithm 3). This mechanism is also used to produce the feature model displayed in the user interface, to provide the combined effect defined in Section 4.4.

To prepare suitable feature models, the inner conflict relation $\otimes_I$ (discussed in Section 5.1.5) applies sets of mutually compatible operations, i.e., Compatible Groups (CGs), to a feature model. The issue is that sets are inherently unordered, but the application order of compound operations is nonetheless significant for producing a

function APPLYCOMPATIBLEGROUP(G, FM, CG)
Require: G is the CDAG for a group of operations GO,
    $FM \in \mathcal{FM}$ is a legal feature model and CG is a compatible group for GO
  $COs \leftarrow$ topological sorting of CG for the causal relationships captured in G
  while $COs.hasNext()$ **do**
    $FM \leftarrow applyCO(FM, COs.next())$
  end while
  return FM
end function

Algorithm 3: Applying a compatible group to a feature model.

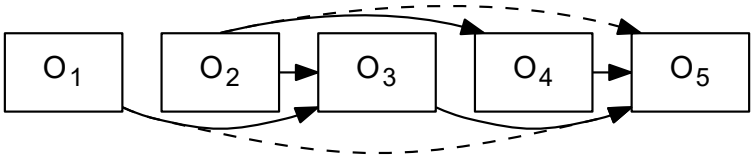


Figure 5.3: Topological sorting of the compatible group shown in Figure 5.1b.

correct result. For example, consider the CDAG depicted in Figure 5.1b and suppose that all operations are mutually compatible. Suppose two more concurrent operations O_6 and O_7 are subsequently generated and checked for conflict. The inner conflict relation will have to apply the set of operations $\{O_1, O_2, O_3, O_4, O_5\}$ to produce a feature model on which O_6 and O_7 may be applied. However, it is not clear in which order to apply these operations, as different orders may lead to different results. For instance, if O_1 sets a feature to mandatory and O_3 undoes the change by setting it to optional, O_1 must be executed before O_3 to respect the causal ordering of operations. We infer that a suitable application order has to respect all causal relationships captured by the group of operation's CDAG.

Thus, we propose a *topological sorting strategy* where the application order of operations in a compatible group is a topological sorting of the operations according to their causal relationships. A *topological sorting* is a linear ordering of operations that respects the causally-preceding relation, i.e., if $O_a \rightarrow O_b$, O_a comes before O_b in the topological sorting. A topological sorting exists for any strict partial order, such as the causally-preceding relation, and can be algorithmically constructed [Kah62]. For example, suppose the compatible group shown in Figure 5.1b should be applied to a feature model. To produce the correct feature model, applyCompatibleGroup first determines a topological sorting of these operations. One possible topological sorting is O_1, O_2, O_3, O_4, O_5 as can be seen in Figure 5.3, read from left to right.

However, the topological sorting for a compatible group is not necessarily unique. For example, in Figure 5.3, O_1 and O_2 may be swapped to create another topological sorting which applyCompatibleGroup might use. In general, other topological sortings can be obtained by swapping concurrent operations. In Section 5.4, we show that this does not impact the resulting feature model, i.e., applyCompatibleGroup is stable across sites although multiple topological orderings may exist. In the next section, we use this topological sorting strategy to devise the inner conflict relation.

5.1.5 Inner Conflict Relation

The *inner conflict relation* $\otimes_I$ is responsible for the construction of an appropriate targeted feature model and detection of conflicts specific to feature modeling. To detect conflicts between compound feature modeling operations, we introduce Algorithm 4, syntacticallyConflicting. For two given compound operations CO_a and CO_b in a group of operations, it determines whether they have a *syntactic conflict*, i.e., a conflict that concerns basic syntactic properties of feature models. The basic strategy of the algorithm is as follows: First, it applies one compound operation, CO_y, to an appropriate feature model. Then, it subsequently applies the POs that CO_x contains to this feature model, so that in the end, both CO_x and CO_y

are applied to the feature model. While applying CO_x's POs, the current feature model can be inspected for potential consistency problems using a set of *conflict detection rules* discussed below. If any inconsistency is found, the algorithm reports a conflict.

Besides the checked operations, the algorithm additionally requires the group of operations' CDAG and its base feature model as arguments. The *base feature model* for a CDAG is its initial feature model, to which no operations from the CDAG have been applied yet. Every operation in the CDAG is based on this feature model, therefore it can be considered the operations' common ancestor. From the base feature model, the topological sorting strategy applyCompatibleGroup can produce feature models that correspond to each operation in the CDAG.

When run, syntacticallyConflicting first prepares several feature models which are required for conflict detection. In Section 5.1.3, we stated that an operation's execution context should match its generation context. The generation context of a CO corresponds to the feature model that includes the effects of all of its causally preceding operations. Thus, it can be obtained by applying all operations in the CO's CP set to the base feature model with applyCompatibleGroup. The algorithm uses this technique to prepare three feature models: $FM_{prec\,CO_y}$ includes all operations that causally precede CO_y, but not CO_y itself, while FM_{CO_y} additionally includes CO_y. These feature models are required by some of the conflict detection rules discussed below. Finally, FM further includes all operations that causally precede CO_x, but not CO_x itself. In this step, operations that precede both CO_x and CO_y are excluded, because they have already been applied to $FM_{prec\,CO_y}$.

FM is then subsequently modified by applying all POs in CO_x in order. For a given PO in CO_x, FM contains CO_y (and all of its preceding operations), all preceding operations of CO_x, and all previous POs contained in CO_x. Because FM contains CO_y and all previous POs of CO_x, it can be used to detect any consistency problems caused by the simultaneous application of both operations. To this end, we introduce a set of *conflict detection rules*, each of which inspects whether the current PO's metadata is allowed in the context of FM. If any rule applies for any PO, a conflict has been detected and syntacticallyConflicting returns *true*. Otherwise, it returns *false*, and the checked operations are considered compatible. The rules are applied in order and return early, so that every rule may assume that the previous rules have not detected a conflict. We proceed to explain the individual rules.

- **No-Overwrites Rule**. This rule resembles the original GRACE conflict relation (cf. Definition 4.1) in that no two concurrent operations may update the same feature's or cross-tree constraint's attribute to a new value. For this, an update PO's *oldValue* can be used, which corresponds to the attribute's value in the PO's generation context. If the current attribute value does not match *oldValue*, an operation included in FM must have changed it. Because $\otimes_O$ guarantees that all causally preceding operations are compatible with CO_x, CO_y must have updated the attribute, therefore CO_x and CO_y are in conflict. Apart from preserving user intentions, this rule is needed by the *createFeatureAbove* CO, which requires that all supplied features are siblings in the CO's generation and execution context. This is guaranteed by this rule because

function SYNTACTICALLYCONFLICTING(G, FM_{base}, CO_x, CO_y)
Require: G is the CDAG for a group of operations GO,
FM_{base} is the base feature model for G, and $CO_x, CO_y \in GO$
$FM_{precCO_y} \leftarrow$ APPLYCOMPATIBLEGROUP($G, FM_{base}, CP_G(CO_y)$)
$FM_{CO_y} \leftarrow$ 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
$PO_x = updateFeaturePO(F^{ID}, attribute, oldValue, newValue)$
$\wedge\ oldValue \neq FM.\mathcal{F}(F^{ID}).[attribute]$
▷ **No-Overwrites Rule** for cross-tree constraints
$\vee\ PO_x = updateConstraintPO(C^{ID}, attribute, oldValue, newValue)$
$\wedge\ oldValue \neq FM.\mathcal{C}(C^{ID}).[attribute]$
▷ **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
$\vee\ PO_x = updateFeaturePO(F^{ID}, attribute, oldValue, newValue)$
$\wedge\ F^{ID} \in \mathcal{F}^{\dagger}_{FM}$
$\wedge\ \neg(attribute = parentID \wedge oldValue = \dagger)$
▷ **No-Graveyarded Rule** for parent features
$\vee\ PO_x = updateFeaturePO(F^{ID}, attribute, oldValue, newValue)$
$\wedge\ attribute = parentID$
$\wedge\ newValue \in \mathcal{F}^{\dagger}_{FM}$
▷ **No-Graveyarded Rule** for cross-tree constraints
$\vee\ PO_x = updateConstraintPO(C^{ID}, attribute, oldValue, newValue)$
$\wedge\ C^{ID} \in \mathcal{C}^{\dagger}_{FM}$
$\wedge\ \neg(attribute = \dagger \wedge oldValue = true)$
▷ **No-Group-Optional Rule**
$\vee\ PO_x = updateFeaturePO(F^{ID}, attribute, oldValue, newValue)$
$\wedge\ attribute = optional$
$\wedge\ (FM.\mathcal{F}(F^{ID}).parentID = \bot$
$\vee\ FM.\mathcal{F}(FM.\mathcal{F}(F^{ID}).parentID).groupType \neq and)$
▷ **Assert-No-Child-Added Rule**
$\vee\ PO_x = assertNoChildAddedPO(F^{ID})$
$\wedge\ \{FC^{ID} \in \mathcal{F}^{ID}_{FM_{CO_y}} \mid FM_{CO_y}.\mathcal{F}(FC^{ID}).parentID = F^{ID}\} \setminus$
$\{FC^{ID} \in \mathcal{F}^{ID}_{FM_{precCO_y}} \mid FM_{precCO_y}.\mathcal{F}(FC^{ID}).parentID = F^{ID}\} \neq \varnothing$
then return *true*
$FM \leftarrow$ APPLYPO(FM, PO_x)
end while
return *false*
end function

Algorithm 4: Detecting syntactic conflicts between feature modeling operations.

concurrent updates on the supplied features' *parentID*s would cause a conflict with CO_x. Similarly, the *removeFeature* CO requires that the removed feature's *parentID* is equal in its generation and execution context, which is also guaranteed by this rule.

- **No-Cycles Rule**. As per Definition 3.3, legal feature models may not contain any cycles. This rule flags CO_x and CO_y as conflicting when a *parentID* update PO in CO_x moves a feature below a feature that is, in turn, moved by CO_y. This is checked by inspecting the move target's path to $\perp$ or † as captured by the *descends from* relation. This rule handles the above-mentioned example where concurrent *moveFeatureSubtree* operations move feature subtrees below each other. This phrasing of the rule also detects arbitrary *n-wise conflicts*, i.e., a group of n conflicting operations that only introduces a cycle when all n operations are applied together (comparable to the *feature interaction* problem [BDC+89, CKMRM03]). For example, suppose a *moveFeatureSubtree* operation moves the feature `A` below `B`, a second moves `B` below `C` and a third moves `C` below `A`: The cycle only emerges when all three operations are applied together, not individually or pairwise. As the MOVIC algorithm only supports pairwise conflict detection, we phrase this rule in a generic way that also detects n-wise conflicts, although this may introduce some false positives.
- **No-Graveyarded Rule**. This rule guarantees that no operation can target a graveyarded feature or cross-tree constraint. For example, this may happen if one collaborator removes a feature or feature subtree and another collaborator concurrently sets an attribute on one of those features. In addition, setting a feature parent or a cross-tree constraint formula may conflict with a removal operation when one of the removed features is referenced. The rule checks whether a feature or cross-tree constraint targeted by CO_x is graveyarded by CO_y, in which case they are flagged as conflicting. There is one exception to this rule, namely when an already graveyarded feature or cross-tree constraint is removed from the graveyard, i.e., a removal operation is undone.
- **No-Group-Optional Rule**. In feature models, *alternative* and *or* groups imply that children features have no *optional* attribute. In Section 3.1, we explained that we still save the *optional* attribute of such children features, but that it is ignored by the user interface. However, this still allows one collaborator to set a feature with an *and* group to an *alternative* group, while another collaborator concurrently sets a child feature to optional or mandatory. We introduce this rule to detect such conflicts by checking whether an *optional* update PO targets a feature that is part of an *alternative* or *or* group. Further, it forbids that the *optional* attribute on the root feature is set, because the root feature is always mandatory.
- **Assert-No-Child-Added Rule**. In the *removeFeature* compound operation, a single feature is removed after pulling up all child features one level. Because a *removeFeature* CO contains the pulled-up child features, the child features must be the same in the CO's generation and execution context. In other words, no child features may be removed or added concurrently for the removed feature. Concurrent removal of child features is already handled by the *no-overwrites* rule, because this would change the removed child features'

*parentID*s and therefore conflict with CO_x. However, adding child features concurrently is still allowed. We introduce this rule to forbid that any child features are added to a feature by CO_y, so that the *removeFeature* CO's execution context is always valid. It does so by checking whether there is any new child feature after, but not before applying CO_y by inspecting the child features in FM_{CO_y} and FM_{precCO_y}, respectively.

This set of conflict detection rules preserves some basic consistency properties of feature models, as well as user intentions. As for the consistency properties, the rules help to preserve the legality of feature models (defined in Section 3.1) also in the presence of concurrent operations, therefore extending Theorem 3.1: The feature model versions created when using these rules are always acyclic, as guaranteed by the *no-cycles* rule. Further, the resulting feature models still have a single root feature: This condition can only be violated by a *removeFeature* CO on the root feature, which may only be generated when the root feature has exactly one child (cf. Section 3.2). Together, the *no-overwrites* and *assert-no-child-added* rules ensure that a removed feature's children are not updated by concurrent operations. This also applies to the root feature, therefore applying a root removal CO will always result in a single new root feature.

Regarding user intentions, these rules avoid surprises in different ways: First, the *no-overwrites* rule detects if two collaborators concurrently set the same feature's or cross-tree constraint's attribute. Thus, no updates are ever lost or overridden. Second, the *no-graveyarded* rule detects whether a collaborator updates a feature or cross-tree constraint which has been graveyarded by another collaborator concurrently. This way, no removal operation can override other collaborators' update operations. Finally, the *no-group-optional* rule detects when *groupType* updates would override concurrent updates of *optional* attributes; further, it ensures that the root feature is always mandatory. Thus, this rule prevents that *optional* attribute updates are overridden.

To determine $\otimes_I$ for two given compound operations, we introduce innerConflicting (Algorithm 5), which utilizes syntacticallyConflicting to check COs for conflict. That is, innerConflicting first checks whether subsequently applying CO_a's POs after applying CO_b results in a conflict. Next, it checks the same vice versa, first applying CO_a and then subsequently POs from CO_b. This is necessary because both CO_a and CO_b have to be applied subsequently against the other operation, so that all conflicts can be detected. For example, if CO_a removes a feature and CO_b updates the same feature's *optional* attribute, only the second call to syntacticallyConflicting will detect the conflict, as it applies CO_b subsequently against CO_a so that the *no-graveyarded* rule is triggered as expected.

If both calls to syntacticallyConflicting did not detect a conflict, innerConflicting may check additional arbitrary semantic properties on feature models. A *semantic property* on feature models is a deterministic function $SP\colon \mathcal{FM} \to \{true, false\}$ that returns whether a given legal feature model includes a semantic inconsistency. The set of all semantic properties that should be checked by innerConflicting is denoted as $\mathcal{SP}$. The $\mathcal{SP}$ set may be adjusted according to the collaborators' needs when the system is initialized. For example, collaborators

function innerConflicting(G, FM_{base}, CO_a, CO_b)
Require: G is the CDAG for a group of operations GO,
FM_{base} is the base feature model for G, and $CO_a, CO_b \in GO$
if $CO_a = CO_b$ **then return** *false*
if syntacticallyConflicting(G, FM_{base}, CO_a, CO_b) **then return** *true*
if syntacticallyConflicting(G, FM_{base}, CO_b, CO_a) **then return** *true*
▷ check additional semantic properties
$FM \leftarrow$ applyCompatibleGroup(G, FM_{base},
$CP_G(CO_a) \cup CP_G(CO_b) \cup \{CO_a, CO_b\}$)
if $\exists SP \in \mathcal{SP}$: SP($FM$) = *true* **then return** *true*
return *false*
end function

Algorithm 5: Computing the inner conflict relation $\otimes_I$ for compound operations.

may choose to ensure some or all of the following well-known semantic properties discussed in Section 2.1.2: The modeled SPL must always have at least one product (*void feature model* analysis); the feature model may not include *dead* or *false-optional features*; or the feature model may not include *redundant cross-tree constraints* [ABKS13, BSRC10, FBGR13]. These properties are useful to reason not only about the feature model, but also the products it represents. Because the associated analysis algorithms may be slow on large feature models [PLP11], collaborators may also choose to not check any semantic properties to improve responsiveness, i.e., $\mathcal{SP} = \varnothing$.

Note that the MOVIC algorithm only allows pairwise detection of conflicts, therefore any inconsistencies that arise from *n-wise conflicts* as mentioned above are not guaranteed to be detected. However, we argue that such n-wise conflicts are rare and difficult to detect (as noted in the context of feature interactions [ARW+13]) and can therefore be disregarded. In the unlikely case that such a conflict occurs, the system may still analyze the feature model in regular intervals and notify the collaborators about the problem, so that they can fix it manually without assistance from the MOVIC algorithm. Further, this problem is only relevant when pairwise conflicts can actually occur for a semantic property: For example, a simple semantic property that guarantees *unique feature names* has to cover pairwise conflicts only.

To check the chosen semantic properties, applyCompatibleGroup is used to prepare a feature model that includes the execution effects of both CO_a and CO_b, as well as all their causally preceding operations. The semantic properties in $\mathcal{SP}$ can then be checked in any order with respect to this feature model. In addition, each semantic property in $\mathcal{SP}$ must be checked when an operation is generated, to ensure that operations are valid individually. If no semantic property detected an inconsistency (or $\mathcal{SP} = \varnothing$), CO_a and CO_b are considered compatible. Using innerConflicting, $\otimes_I$ can then be computed as follows:

Definition 5.5. *Two compound operations CO_a and CO_b are in* inner conflict, *i.e., $CO_a \otimes_I CO_b$, if and only if* innerConflicting(G, FM_{base}, CO_a, CO_b) = *true, where G and FM_{base} are the current CDAG and base feature model at the site that executes* innerConflicting.

In Section 5.4, we show that $\otimes_I$ is a conflict relation and can be used in conjunction with $\otimes_O$ to guarantee convergence and intention preservation.

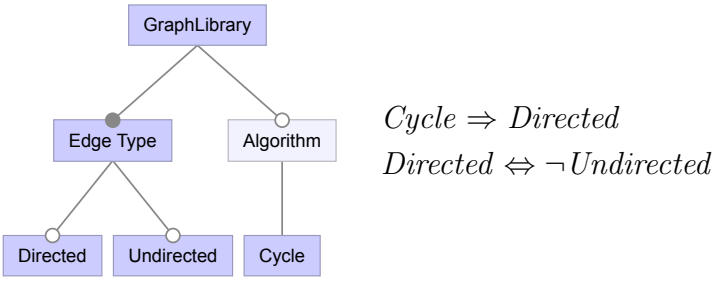
5.1.6 An Integrated Example

Before we move on to discussing our approach for conflict resolution, we give an example that integrates all mechanisms for detecting conflicting operations discussed above. Suppose two collaborators A and B are editing the feature model shown in Figure 5.4a which is an excerpt of the graph product line described in Section 2.1.1. Each collaborator generates two operations on this feature model (shown in Figure 5.4b–5.4e). However, at site A, a temporary network outage prevents that any operations are sent and received. Thus, there is a short time window where A and B are oblivious of the other's operations, i.e., their operations are concurrent.

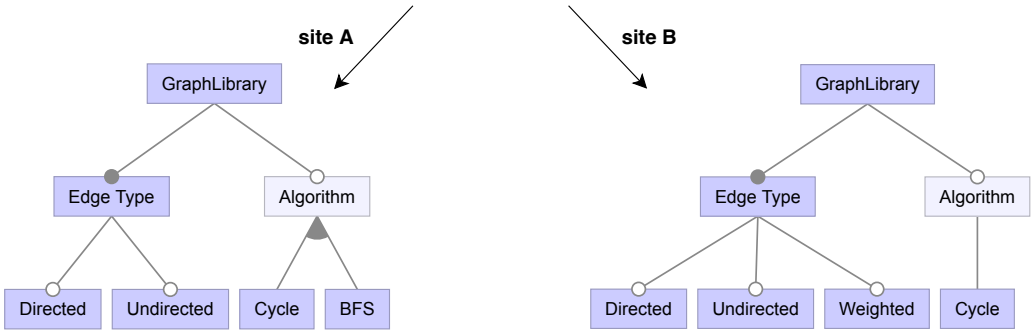
At site A, the collaborator's intention is to add variability to the modeled product line by adding a `BFS` feature below `Algorithm` that computes a breadth-first search on a graph (O_1). After creating the feature, she notices that the feature model contains a small problem: The `Edge Type` feature is concrete, although it is only used to structure the feature tree. She comes to the conclusion that the `Edge Type` feature is not really necessary, and removes it, pulling up its child features in the process (O_3). Meanwhile, the collaborator at site B also intends to add variability by adding a feature. He adds the `Weighted` feature below `Edge Type`, which is used to represent graphs with weighted edges (O_2). Afterwards, he also notices the problem regarding `Edge Type`, but solves it differently than the other collaborator, i.e., by making it an abstract feature (O_4). The resulting feature models at site A and B are shown in Figure 5.4d and Figure 5.4e, respectively.

As soon as site A has regained its network connection, all pending operations are exchanged between both collaborators. At site A, O_2 and O_4 arrive in that order because $O_2 \rightarrow O_4$ (guaranteed by a causality preservation scheme as discussed in Section 5.4). Similarly, O_1 and O_3 arrive at site B in that order. In Figure 5.5, we show the CDAG G at both sites after they have received all pending operations. We now investigate how the MOVIC algorithm accommodates these operations into the MCGS shown in Figure 5.4f. For that, we need to determine the conflict relationships between the involved operations according to $\otimes_O$, which MOVIC invokes:

- Only concurrent operations can conflict, therefore $O_1 \odot_O O_3$ and $O_2 \odot_O O_4$.
- For checking O_1 and O_2, $\otimes_O$ first checks all causally preceding operations. Because $CP_G(O_1) = CP_G(O_2) = \varnothing$, $\otimes_O$ immediately invokes $\otimes_I$ to check O_1 and O_2 for inner conflict. $\otimes_O$ applies O_1 and O_2 in both orders to the base feature model (Figure 5.4a) and checks whether any included PO violates a conflict detection rule. Because both O_1 and O_2 only create and update a new feature, they cannot conflict in any way. Thus, $\otimes_I$ (and $\otimes_O$) conclude that they are compatible, i.e., $O_1 \odot_O O_2$.
- To check O_1 and O_4, $\otimes_O$ first checks whether $O_1 \odot_O O_2$, which is the case. Thus, $\otimes_I$ is invoked, which applies O_1 and O_4 to the feature model shown in Figure 5.4c. Because O_1 creates a new feature and O_4 sets another feature's abstract attribute, they are compatible as well, i.e., $O_1 \odot_O O_4$.

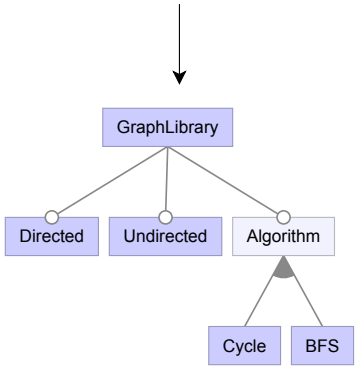


(a) Base feature model (constraints omitted below)

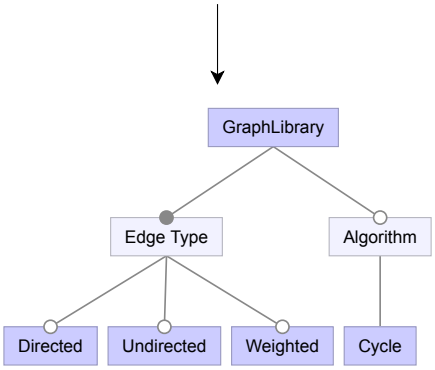


(b) $O_1 = createFeatureBelow(BFS, Algorithm)$

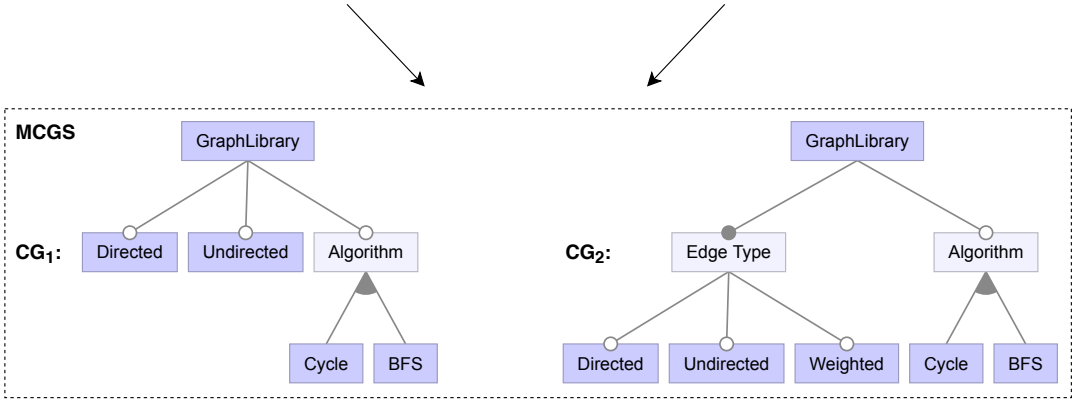
(c) $O_2 = createFeatureBelow(Weighted, EdgeType)$



(d) $O_3 = removeFeature(EdgeType)$



(e) $O_4 = setFeatureAbstract(EdgeType, true)$



(f) $MCGS = \{CG_1, CG_2\}$ where $CG_1 = \{O_1, O_3\}$ and $CG_2 = \{O_1, O_2, O_4\}$

Figure 5.4: Detecting conflicts for concurrent compound operations. The *FM* parameter has been omitted for brevity, and we label features with their IDs.

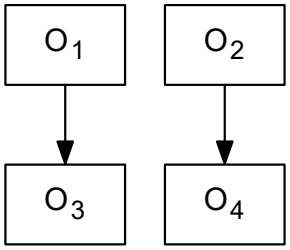


Figure 5.5: CDAG for the compound operations shown in Figure 5.4.

$$
\begin{aligned}
O_1 = [&\mathit{createFeaturePO}(\mathit{BFS}),\\
&\mathit{updateFeaturePO}(\mathit{BFS}, \mathit{parentID}, \dagger, \mathit{Algorithm})]\\
O_2 = [&\mathit{createFeaturePO}(\mathit{Weighted}),\\
&\mathit{updateFeaturePO}(\mathit{Weighted}, \mathit{parentID}, \dagger, \mathit{EdgeType})] \qquad \triangleright\ PO_2\\
O_3 = [&\mathit{assertNoChildAddedPO}(\mathit{EdgeType}),\\
&\mathit{updateFeaturePO}(\mathit{Directed}, \mathit{parentID}, \mathit{EdgeType}, \mathit{GraphLibrary}),\\
&\mathit{updateFeaturePO}(\mathit{Undirected}, \mathit{parentID}, \mathit{EdgeType}, \mathit{GraphLibrary}),\\
&\mathit{updateFeaturePO}(\mathit{EdgeType}, \mathit{parentID}, \mathit{GraphLibrary}, \dagger)]\\
O_4 = [&\mathit{updateFeaturePO}(\mathit{EdgeType}, \mathit{abstract}, \mathit{false}, \mathit{true})]
\end{aligned}
$$

Figure 5.6: Decomposition of the operations in Figure 5.4 into primitive operations.

- When checking O_2 and O_3, $\otimes_O$ again checks whether $O_1 \odot_O O_2$, which is true. $\otimes_I$ therefore applies O_2 and O_3 to the feature model shown in Figure 5.4b. During the check, it applies O_3 and then subsequently the POs contained in O_2 (shown in Figure 5.6). However, when SYNTACTICALLYCONFLICTING encounters PO_2, the *no-graveyarded* rule for parent features applies because $\mathit{EdgeType} \in \mathcal{F}^{\dagger}_{FM}$. Thus, O_2 and O_3 are in conflict ($O_2 \otimes_O O_3$) because O_2 tries to create a feature below `Edge Type`, which is graveyarded by O_3.
- To check O_3 and O_4, $\otimes_O$ requires (among others) that all causally preceding operations for O_4 are compatible with O_3. However, $O_2 \otimes_O O_3$, therefore $O_3 \otimes_O O_4$ by the transitivity property because there is no meaningful feature model to which O_3 and O_4 may be applied together.

We find that O_2 and O_3 are in conflict (and therefore also O_3 and O_4), while all other operations are compatible. In Example 4.1, we have already seen how the MOVIC algorithm constructs an MCGS for operations with these conflict relationships. Thus, at both sites the resulting MCGS consists of two MCGs, $CG_1 = \{O_1, O_3\}$ and $CG_2 = \{O_1, O_2, O_4\}$.

At this point, both collaborators need to decide on an MCG with which they would like to continue, i.e., they have to resolve their conflict. To support them in their decision, we may use the combined effect defined in Section 4.4 to construct the feature models corresponding to the MCGs (as depicted in Figure 5.4f). This is possible because we consider only one global targeted object, the feature model

(cf. Section 5.1.1). To construct these feature models, we employ the topological sorting strategy introduced in Section 5.1.4. For example, to produce the feature model associated with CG_2, applyCompatibleGroup is invoked with the base feature model and the set of compatible operations that should be applied, i.e., $\{O_1, O_2, O_4\}$. applyCompatibleGroup then determines a topological sorting of these operations that respects the causally-preceding operation. In this case, two possible topological sortings exist: O_1, O_2 and then O_4; or O_2, O_4 and then O_1. applyCompatibleGroup applies any of these topological sortings to the base feature model and arrives at the model shown in Figure 5.4f as CG_2.

5.2 Conflict Resolution

Our approach, as described so far, fully automates the detection of conflicts and allocation of feature model versions using the MOVIC algorithm. This algorithm, however, only offers functionality for constructing an MCGS and its associated feature models, not for resolving conflicts. *Conflict resolution* is the process where, in case of conflict, collaborators examine alternative feature model versions and negotiate a specific version [SFB+87, Wul95]. In Section 4.3, we argued that manual conflict resolution is reasonable for collaborative feature modeling in order to ensure intention preservation and avoid surprising outcomes. In this section, we propose a simple manual conflict resolution process (depicted in Figure 5.7) that allows collaborators to cast *votes* for their preferred feature model versions. We choose a voting technique because it is flexible (as described below) and emphasizes fairness [DPN98, Gib89, MRS+04].

Site Freeze

First, we assume that as soon as a conflict is detected, a site switches to conflict resolution mode and forbids any further editing, referred to as a *site freeze*. We make this decision for several reasons: First, it is not clear which feature model version should be edited when a conflict has been detected. Possibly, the user may target one particular version, which would promote divergence of the versions; or all versions at once, which is difficult to communicate in a user interface. Second, it forces collaborators to address the conflict, therefore avoiding any further divergence. This corresponds to the targeted use case of collaborative real-time editors: Tight synchronization and early conflict resolution. For asynchronous workflows involving branching and divergence, collaborators should consider using a version control system instead. Third, the MOVIC algorithm has been proven only for operations that target the same object version [SC02]. In particular, it does not guarantee convergence when operations explicitly target versions created during the algorithm [XOZ03]. Xue et al. [XOZ03] introduce conflict relations for solving this problem, but their approach is tailored to the GRACE system. Instead, freezing a site on conflict is also a solution to ensure the convergence property, which we expand upon in Section 5.4. As we expect conflicts to be rare, this is a reasonable approach.

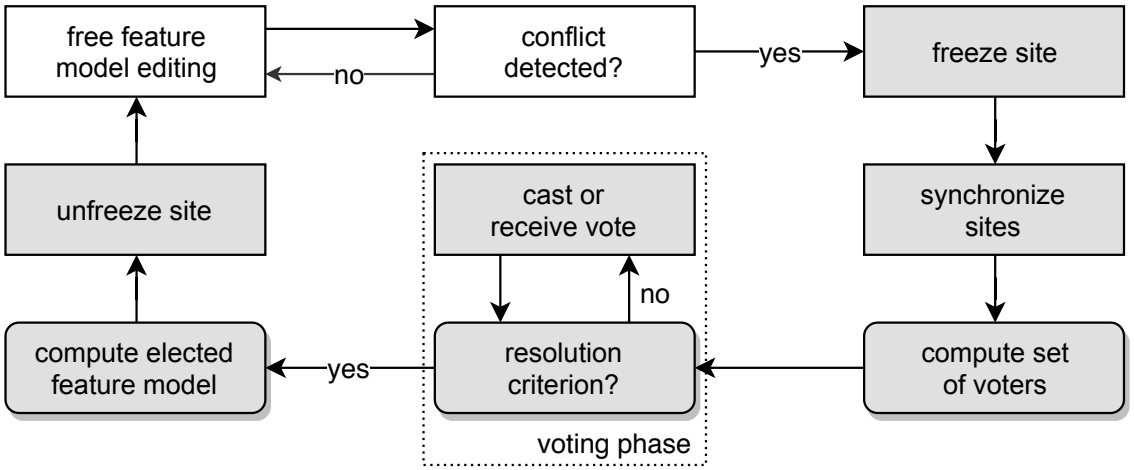


Figure 5.7: Manual conflict resolution process. The process comprises all highlighted steps. For the steps in the third row, we discuss several concrete techniques.

Conflict Resolution Process

A site switches to conflict resolution mode (freezing the site in the process) as soon as it learns about a conflict, i.e., the MCGS contains more than one MCG. The collaborator is then presented with a suitable user interface that allows her to compare and investigate the conflicting operations. To make sure that a collaborator has seen all versions available for voting, the system does not allow voting until all unseen operations from other sites have been received. This is guaranteed to happen, because all other sites will be gradually frozen due to the conflict, too. We argue that this brief synchronization period is inevitable, because the collaborator must be able to make an informed decision. Also, collaborators may not even notice the synchronization period, as they need a few moments to examine the alternative versions before making a decision. When all sites are synchronized, every site computes a set of *voters* V that determines the collaborators eligible to vote. This set may be computed independently from other sites, as long it is equal across all sites. How V should be determined depends on the collaborators' preferences; below, we discuss several strategies. Further, a set of *vote results* VR is initialized to an empty set.

Now, the *voting phase* starts: Every voter in V may cast a *vote* on a single feature model version, which stores the voter and the elected version. This vote is added to the local VR set and propagated to all other sites. When a site casts or receives a vote, it simply adds it to its local VR set. Note that once cast, a vote is final and cannot be taken back. This allows us to keep the voting phase entirely optimistic; in fact, VR corresponds to a *grow-only set* CRDT (cf. Section 4.2), which provable converges [SPBZ11a]. Whenever a site processes a vote, it checks a *resolution criterion* that decides whether the voting phase is complete. Similar to the set of voters V, this resolution criterion should come to a consistent conclusion across sites. Below, we discuss a number of different resolution criteria, one of which may be chosen according to the collaborators' preferences. As soon as the resolution criterion determines that the voting phase is complete, it computes the final elected version CG from the VR set. The voting phase is now complete, so that the site may remove

all versions but CG from the MCGS and unfreeze the site. To conclude the conflict resolution process, all operations not included in CG are removed from the CDAG, as if they had never been submitted. Note that the entire voting phase does not require synchronization. Thus, a collaborator can immediately continue editing as soon as the resolution criterion is satisfied locally, although other sites may still be frozen due to network latencies.

Neutral Compatible Group

In some instances, collaborators may want to abort the resolution process and return to a state before any conflicting operations were submitted. For example, this may be the case when the collaborators fail to agree upon a final elected version. Simply resetting the MCGS to an empty set does not work as intended, because this would cancel all operations applied to the base feature model, including compatible operations which are included in all CGs. Instead, we introduce an additional *neutral* CG that may be chosen to cancel the conflict resolution. This CG is computed from the MCGS just before the voting phase starts and added to the user interface as an alternative version. It is defined as the intersection of all CGs, i.e., $CG_{neutral} \coloneqq \bigcap_{CG \in MCGS} CG$. Thus, it contains all operations which are applied in every feature model version and have no conflict potential with any other operations. If the resolution criterion settles on the neutral CG at the end of the voting phase, the system is therefore restored to a state without any conflicting operations. In Section 5.4, we show that the neutral CG is well-defined and yields the expected result.

Voting Strategies

Above, we have introduced a voting technique to determine a final elected feature model version. This technique can be adapted flexibly to the collaborators' preferences with different voting strategies. In particular, the set of voters V and the resolution criterion can be adjusted in different ways, which we discuss now.

The set of voters V may be adjusted to restrict which collaborators may cast a vote. For example, this is useful if only a subset of collaborators can competently evaluate the alternative versions, or if some collaborators should have elevated rights. Possible strategies include:

- $V = \varnothing$, i.e., no collaborator may vote. In this case, the resolution criterion is satisfied immediately and must choose a suitable CG automatically (e.g., the neutral CG). This strategy completely avoids manual conflict resolution and rejects any conflicting operations, thus violating our intention preservation requirement (cf. Section 4.1). However, it is very simple to implement because it does not require a user interface for conflict resolution.
- V is a fixed set including a single or few collaborators. This may be desirable in environments where collaborators frequently disagree, or where a few collaborators should have elevated rights (e.g., a moderator).
- V consists of all collaborators that are involved in the conflict. A collaborator is involved if she submitted an operation that is not included in all CGs, i.e.,

conflicts with some other operation. This strategy is reasonable because the collaborators responsible for the conflict are expected to be most capable of resolving it.

- V consists of all collaborators in the system. This strategy has the advantage of fairness, as no collaborator is excluded from the resolution. However, not every collaborator may be able to judge which version is most suitable as they are not necessarily involved in the conflict.

These strategies can also be combined: For example, when not only all involved collaborators, but also collaborators with elevated rights may vote.

The resolution criterion may be adjusted as well to influence the outcome of the conflict resolution process. Some possible resolution criteria include:

- The voting phase ends when $|V| = |VR|$, i.e., all votes have been cast. The final elected version is the CG with the most votes. This is a simple strategy, but many collaborators may still disagree with the chosen version.
- The voting phase ends when $|V| = |VR|$. If the CG with the most votes has the absolute majority, it is the final elected version. Otherwise, the neutral CG is chosen. This way, the interests of the majority of collaborators are respected.
- The voting phase ends when $|V| = |VR|$ or VR includes at least two votes for different versions. If all collaborators reached consensus (i.e., all votes were cast for the same version), the agreed CG is the final elected version. Otherwise, the neutral CG is chosen. This strategy strictly enforces that all collaborators agree upon one version. This may be unrealistic when V consists of all collaborators, but reasonable if only collaborators involved in the conflict may vote. Further, the voting phase may be concluded early when it is clear that consensus cannot be reached anymore, therefore accelerating the resolution process.

All strategies for the set of voters and all conflict criteria proposed above come to the same conclusion across sites and are therefore suitable for use in the described conflict resolution process.

5.3 Garbage Collection

In the previous sections, we introduced several algorithms to provide for responsive, optimistic concurrency control in collaborative feature modeling. These algorithms rely on two primary data structures, the CDAG and the MCGS: The MOVIC algorithm, which constructs the MCGS; further, the incremental construction of the CDAG; as well as OUTERCONFLICTING, SYNTACTICALLYCONFLICTING, and INNERCONFLICTING, which depend on the CP and CIP sets. However, currently these data structures grow unbounded over time, because each processed operation is added to the CDAG and at least one MCG. Consequently, without further measures, the performance of our system deteriorates over time, which diminishes responsiveness.

To keep the system performant, we introduce a *garbage collection* scheme that periodically prunes the MCGS and CDAG by removing operations that are no longer needed by conflict detection or resolution [Che01, SJZ+98]. Such operations are characterized by the following definition.

Definition 5.6. *Let GO be a group of operations. A compound operation* $CO_a \in GO$ *is* eligible for garbage collection *if and only if for every site in the system, there is a* $CO_b \in GO$ *generated by that site such that* $CO_a \rightarrow CO_b$.

Thus, an operation may be garbage collected when it has been causally succeeded by every site in the system. This works as intended for two reasons: First, an eligible operation cannot be concurrent to any newly processed operation, therefore it cannot provoke any new conflict. Second, such an operation is no longer required by conflict resolution: Any operation involved in a conflict can only be causally succeeded by all sites after the system has been unfrozen, i.e., when conflict resolution has been concluded. Thus, this garbage collection scheme does not affect the correctness of our concept. Note that it is possible that a site does not send any messages for a longer period of time, for example, when a collaborator only spectates at an editing session. Unfortunately, such a site prevents any operations from being garbage collected. To prevent this, every site should also periodically send a *heartbeat message*, which has no effect on the edited feature model and only informs other sites about the state of a site to facilitate garbage collection [SJZ+98].

Regarding the implementation of such a garbage collection scheme, we refer to Sun et al. [SJZ+98]. We only point out some issues specific to our concept: First, the garbage collection scheme may not be run and no heartbeat messages may be sent while the system is frozen, i.e., during conflict resolution. This avoids garbage collecting operations that are still required and introduces no performance problems, as no operations may be generated at all when the system is frozen. Second, when an operation is garbage collected, the following steps are required: Remove the operation from all MCGs in the MCGS; and remove the operation and all its incident edges from the CDAG. Further, the base feature model (cf. Section 5.1.5) has to be updated to include the garbage collected operations' effects. To this end, we apply all garbage collected operations to the base feature model in any topological order using applyCompatibleGroup. Thus, the base feature model effectively absorbs garbage collected operations.

5.4 Correctness

In Section 5.1–5.3, we described how to extend the MVMD concurrency control technique for collaborative feature modeling. Based on this description, we now present some key insights regarding the correctness of our approach. The goal of this section is to show that our proposed concurrency control approach conforms to the CCI model as described in Section 2.2.2 [SJZ+98]. Ensuring these properties avoids inconsistent feature models, raises the collaborators' confidence in the system and therefore allows effective and efficient editing of feature models. Thus, we aim to satisfy the correctness requirement from Section 4.1. We start with discussing the causality preservation property.

Causality Preservation

A collaborative real-time editing system *preserves causality* when it ensures that for any operation, all causally preceding operations are processed beforehand [SJZ+98]. This property is required for several of the following arguments. In client/server systems with ordered message channels, causality violation cannot occur, therefore causality preservation is achieved automatically [SS99]. Regarding other topologies such as peer-to-peer architectures, suitable causality preservation schemes have been proposed in the literature [Fid88, SJZ+98]. Essentially, such schemes delay the processing of operations artificially until they are *causally ready*. These schemes are not specific to feature modeling and can therefore be applied to our concept without modifications.

In our approach for concurrency control, causality preservation is required to ensure that that for any operation, all causally preceding operations are fully known at all sites. Based on this, we can show that at any site, an operation's CP (and therefore also CIP) set is fully known when the operation is being processed.

Theorem 5.1. *Let G be the CDAG for a group of operations GO and $O_a \in GO$. Further, let G' be the current CDAG at any site at any time. Then, if G' includes O_a, also $CP_{G'}(O_a) = CP_G(O_a)$, i.e., the CP set is computed correctly at said site.*

Proof. Recall that G' is incrementally constructed as a site processes operations. That is, when an operation is generated or received, it is inserted into G' (cf. Section 5.1.2). Due to causality preservation, any $O_b \in CP_G(O_a)$ is processed before O_a at all sites. Thus, when O_a is processed, G' already includes all operations in $CP_G(O_a)$ and $CP_{G'}(O_a) = CP_G(O_a)$. □

Below, we use this property to reason about our proposed conflict relations and the combined effect of an MCGS. We move on to discuss the convergence property.

Convergence

A collaborative feature model 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 [SJZ+98]. To this end, we rely on the correctness of the MOVIC algorithm as described by Sun and Chen [SC02]. Sun and Chen prove that their algorithm converges for operations that target the same object version. We enforce this with our global targeted object strategy (cf. Section 5.1.1) and by freezing all sites when a conflict occurs (cf. Section 5.2). However, in Section 5.1 we introduced two new conflict relations and a topological sorting strategy for use within the MOVIC algorithm, which require further verification to confirm that our system converges.

First, we examine our topological sorting strategy as implemented by applyCompatibleGroup, which applies a set of mutually compatible operations (a Compatible Group) to a feature model. In Section 5.1.4, we mentioned that multiple topological sortings may exist for a Compatible Group. Note that topological sortings of a set of operations only differ in the execution order of concurrent operations. For example, some topological sortings of the set of operations $\{O_1, O_2, O_3, O_4\}$ depicted

in Figure 5.5 include: O_1, O_3, O_2, and then O_4; O_1, O_2, O_3, and then O_4; and O_1, O_2, O_4, and then O_3. Note that these application orders can be obtained by swapping concurrent operations. Further, applyCompatibleGroup requires that the supplied set of operations be mutually compatible. Thus, we only have to show that *compatible concurrent operations* commute in our system. This follows from the *no-overwrites* rule introduced in syntacticallyConflicting: By this rule, compound operations 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 generally commute. Thus, compatible concurrent operations always commute in our system. Consequently, no matter which topological sorting applyCompatibleGroup determines, it computes the same feature model consistently for given parameters. In particular, it is also *stable across sites*, i.e., 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 including subsets of operations' CP sets. Note that a CP set of an operation O_a only includes mutually compatible operations: If the CP set included two conflicting operations O_b and O_c, the system would have been frozen until the conflict had been 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 their union because of the transitivity property of $\otimes_O$ (cf. Definition 5.3). Further, the *no-overwrites* rule has already been checked by syntacticallyConflicting at this point, therefore CO_a and CO_b commute. Because CO_a and CO_b are mutually compatible with their causally preceding operations by definition, the set applied by applyCompatibleGroup altogether only includes mutually compatible operations.
- To display a feature model in the user interface and achieve the combined effect defined in Section 4.4, applyCompatibleGroup is invoked with entire MCGs. By definition, all operations in an MCG are mutually compatible.
- The garbage collection scheme discussed in the previous section invokes applyCompatibleGroup with all garbage collected operations. By definition, garbage collected operations are causally succeeded by every site, i.e., included in the most recent generated operation's CP set at any site. We have already seen that operations in a CP subset are mutually compatible. Thus, this subset of garbage collected operations includes only mutually compatible operations.

Next, we investigate whether our proposed conflict relations $\otimes_I$ and $\otimes_O$ can be used successfully in conjunction with the MOVIC algorithm. According to Xue et al. [XOZ03], both must be irreflexive and symmetric. Further, because we compute both conflict relations algorithmically, the algorithms must be *stable across sites*,

i.e., all sites must always compute the same result for two given operations. First, 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 the transitivity property. 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 5.2. *Recall that $CO_a \otimes_I CO_b$ if and only if* innerConflicting*$(G, FM_{base}, CO_a, CO_b) = true$, where G and FM_{base} are the current CDAG and base 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.*

Proof. First, $\otimes_I$ is stable across sites, i.e., innerConflicting (and syntacticallyConflicting) always return the same results for two given operations CO_a and CO_b, although the passed CDAG and base feature model may differ across sites: The CDAG is only used to determine CP sets, which are fully known for CO_a and CO_b (as shown in Theorem 5.1). The base feature model is only modified by the garbage collection scheme; we have already shown in the previous section that this does not affect correctness. 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.

Next, $\otimes_O$ 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 as shown in Theorem 5.1. 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 the results above and Sun and Chen's original introduction of the MOVIC algorithm, we can derive that our collaborative feature modeling editor converges.

Theorem 5.3. *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 the MOVIC algorithm 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 derived from $MCGS_n$ is identical regardless of the processing order, i.e., the system converges.*

Proof. As per our global targeted object (cf. Section 5.1.1) and site freeze (cf. Section 5.2) strategies, all operations in our system always target a single feature model version. Further, $\otimes_O$ is a conflict relation as shown in Theorem 5.2 and can therefore replace the GRACE conflict relation $\otimes$ in the MOVIC algorithm. Thus, the MOVIC algorithm constructs the same $MCGS_n$ for GO regardless of the processing

order [SC02, Property 3]. The combined effect is then computed by invoking applyCompatibleGroup on the resulting MCGs, which has been shown above to be stable across sites. □

Intention Preservation

A collaborative feature model editing system *preserves intentions* if an operation always has the same execution effect at every site regardless of its execution context, i.e., its generation and execution contexts always match [SJZ+98]. The generation context of an operation consists of the base feature model and all operations that causally precede the operation. The execution context may then additionally include operations that do not conflict with any operations in the generation context. The conflict relations $\otimes_O$ and $\otimes_I$ have been designed from the ground up to ensure this property: $\otimes_O$ only invokes $\otimes_I$ when the transitivity property guarantees that the execution context is appropriate; and $\otimes_I$ uses applyCompatibleGroup to prepare this execution context by applying the checked operations' CG sets.

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

Theorem 5.4. *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 the MOVIC algorithm 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 the MOVIC algorithm, $CP_G(O_a)$ is fully known as shown in Theorem 5.1. 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 transitivity 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, i.e., $CP_G(O_a) \subseteq CG$. □

Furthermore, we desire that no operation is ever rejected, masked or overridden to improve the collaborators' confidence in the system (as discussed in Section 4.1). We ensure this with the *no-overwrites*, *no-graveyarded* and *no-group-optional* rules as discussed in Section 5.1.5.

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

Theorem 5.5. *Let G and $MCGS$ be the CDAG and maximum compatible group set for a group of operations GO. Further, let $CG_{neutral} \coloneqq \bigcap_{CG \in MCGS} CG$ be its* neutral compatible group. *Then, $CG_{neutral}$ is stable across sites. Further, for any $O \in CG_{neutral}$, $CP_G(O) \subseteq CG_{neutral}$.*

Proof. First, $CG_{neutral}$ is stable across sites when conflict resolution is initiated. This is because the $MCGS$ is unique [SC02, Property 1] and converges at all sites (Theorem 5.3). Now, for any $O \in CG_{neutral}$, $O \in CG$ for all $CG \in MCGS$. By Theorem 5.4, also $CP_G(O) \subseteq CG$ for all $CG \in MCGS$, therefore also $CP_G(O) \subseteq CG_{neutral}$. □

From the above discussions and theorems, we infer that our collaborative real-time feature modeling system is *consistent* according to the CCI model defined in Section 2.2.2. As the CCI model has proven itself in practice [SE98, SJZ+98], we are confident that our system allows unconstrained collaboration and high responsiveness, while still ensuring basic consistency properties. We believe that this makes for a balanced editing experience so that collaborators can edit feature models effectively and efficiently.

5.5 Summary

In this chapter, we extended an existing concurrency control technique to allow for collaborative real-time feature modeling. We first discussed how to modify the MOVIC algorithm to allow for detection of conflicts, introducing two conflict relations in the process. Then, we explained how collaborators may resolve conflicts manually using a voting technique. We further discussed how to ensure responsiveness of our system with a garbage collection scheme. Finally, we justified the correctness of our approach by showing that it conforms to an established consistency model.

6. Implementation

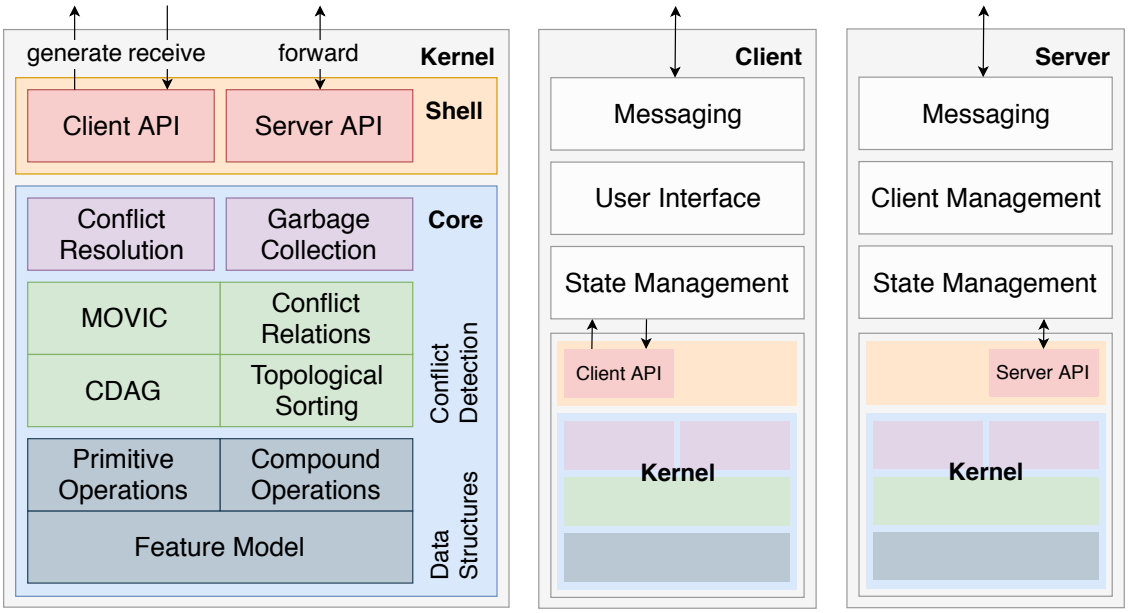
In the previous chapter, we presented the conceptual foundation of a collaborative real-time feature model editor. We now discuss our prototypical implementation of this concept, the Variability Editor (variED), which serves as a proof of concept. We briefly discuss variED's architecture and our choices of technology. Further, we give some directions for other developers regarding future extensions. For more detailed technical documentation, we refer to our GitHub repository[1].

variED is a web-based feature modeling tool, i.e., multiple users may collaborate on a feature model by visiting the same variED instance in their web browser. In this scenario, collaborators assume a client role and connect to a centralized variED server, which acts as a message broker for the clients. Thus, variED employs a client/server architecture (as discussed in Section 2.2.1). Apart from simple integration with the web platform, this has the additional benefit that causality preservation is guaranteed trivially (as discussed in Section 5.4). However, we emphasize that this is only a design choice in our implementation—our concept as such may also be employed in peer-to-peer topologies.

We choose a web-based approach for several reasons: First, it allows for universal and portable usage of our editor across all platforms. This is especially relevant as mobile devices gain more traction, with a global market share of 47.96% for smartphones and 3.83% for tablets in February 2019 [Sta19]. Second, it allows users to view and edit feature models without any setup, e.g., just by visiting a link [GLM11]. This makes our feature modeling platform more accessible to users without a technical background, e.g., domain experts that have no prior experience with feature modeling tools. Third, web browsers are becoming more attractive as an application platform, as they provide sophisticated rendering engines and allow for highly dynamic content [BMG17]. In particular, the high degree of abstraction allowed us to implement our prototype in a relatively short time.

In Figure 6.1, we depict the architecture of variED. We distinguish three components, namely the *collaboration kernel*, *client* and *server* components: The col-

[1] Source code, demo, and documentation available at: https://github.com/ekuiter/variED



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

Figure 6.1: Architecture of variED. Client and server share a collaboration kernel.

laboration kernel (cf. Figure 6.1a) comprises all modules required for collaborative real-time feature modeling. In particular, this includes all concepts and mechanisms discussed in Chapter 3 and 5. We further provide a client and server component (cf. Figure 6.1b–6.1c), which are executed on the client and server sites, respectively. Both components make use of the collaboration kernel through distinct Application Programming Interfaces (APIs). In the following, we discuss each component in more detail.

The collaboration kernel is central to collaborative feature modeling in variED. It is concerned with processing operations; in particular, detecting and accommodating conflicts. For implementation, we choose the *Clojure* programming language [Hic08]. Clojure is a Lisp dialect that provides immutable persistent data structures to allow for a functional programming style. We use Clojure primarily for two reasons: First, it integrates well with our client/server infrastructure, as we discuss below. Second, it allows for a clear programming style free of mutation, which facilitates reasoning about the resulting code. In particular, sets and hash maps are deeply integrated into the language, which allows us to directly translate our pseudocode to Clojure.

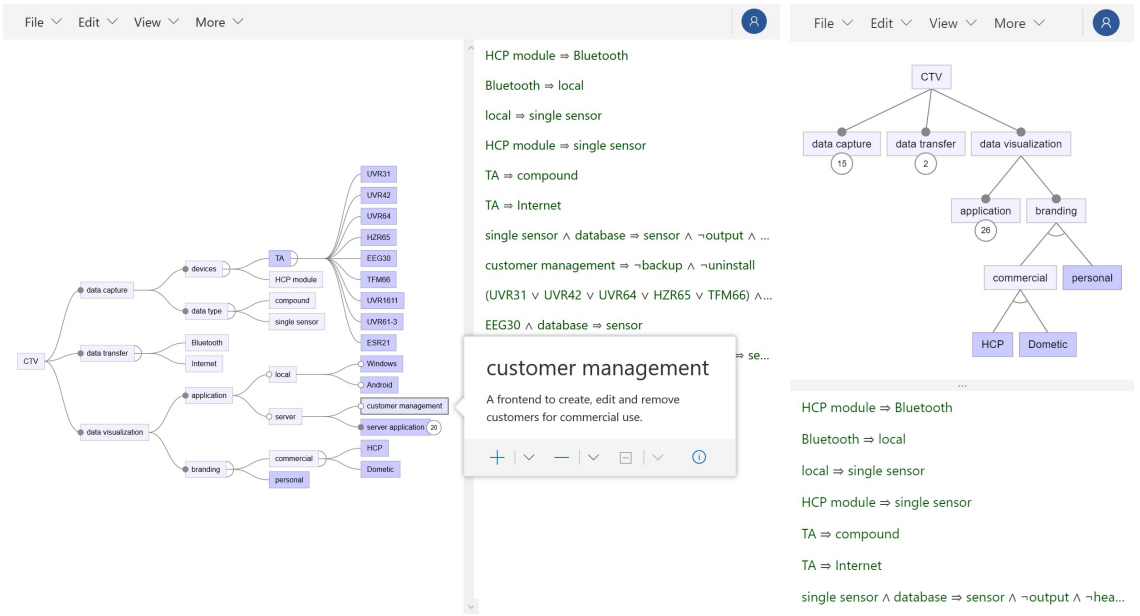
In the collaboration kernel, we employ the *functional core, imperative shell* architecture pattern [Ber12, Coc05]: All business logic for feature modeling as well as detecting and accommodating conflicts is encapsulated in a purely functional *core*. This core is then surrounded by a *shell* of imperative code, which may manipulate site-global data structures such as the current MCGS and CDAG. This separation of concerns facilitates reasoning and testing and keeps the complexity of the imperative code manageable. In Figure 6.1a, we show the simplified functional core, which comprises a module for each concept and mechanism described in this thesis. At the basis of the core, the feature model representation (cf. Section 3.1) and the operation model (cf. Section 3.2) provide the basic data structures for collaborative

feature modeling. Our feature model implementation employs a simple hash table to map from feature and cross-tree constraint IDs to attribute-value pairs. This representation is efficient, simple to maintain and directly corresponds to Definition 3.2. However, other developers may choose another representation, for example when extending an existing tool, such as *FeatureIDE*, with collaboration support. We allow different representations by defining a clear API between the feature model representation and other core modules. We further provide the initial set of primitive and compound operations proposed in Section 3.2, which are open for extension in the future. For conflict detection, we implement the modified MOVIC algorithm with all strategies and mechanisms introduced in Section 5.1. Notably, we further introduce a *conflict cache* that caches the computation results of $\otimes_O$ and $\otimes_I$ to speed up the recursive computation in OUTERCONFLICTING. Note that when extending our concept with more operations, the set of conflict detection rules may need to be adjusted accordingly. Finally, we are currently in the process of integrating our proposed conflict resolution process (cf. Section 5.2) and garbage collection scheme (cf. Section 5.3) into the collaboration kernel. However, to demonstrate the feasibility of our approach, both are not strictly necessary.

The core is employed by the imperative shell (also shown in Figure 6.1a) to allow for actual operation processing. The shell maintains a site-global *context*, which is a global variable comprising the entire state of the collaboration kernel. In particular, the context contains the current MCGS, CDAG, base feature model, and some additional data structures for housekeeping (e.g., the conflict cache). Only few algorithms have access to the site-global context, including the *generate*, *receive*, and *forward* algorithms that comprise the collaboration kernel's external API. *Generate* is called at a client whenever the user issues an editing operation. According to our optimism requirement (cf. Section 4.1), *generate* immediately applies the operation and then processes it locally by adding it to the CDAG and MCGS. The data returned by *generate* is intended to be sent to the server to serve as input to *forward*. The *forward* algorithm (only required at the server) retrieves an operation, processes it according to the MOVIC algorithm and returns it again, intended to be forwarded to all clients but the original issuing client. Finally, *receive* accepts such a forwarded operation at the client by processing and applying it to the feature model, possibly creating multiple versions in the process.

To ensure unique identifiers across sites, we utilize randomly generated UUIDs, which have very low probability for collision [LMS05]. Further, to track the causal ordering relation introduced in Section 2.2.2, we employ the *vector clock* algorithm independently developed by Fidge [Fid88] and Mattern [Mat88]. With this information, we can insert operations into the CDAG and subsequently retrieve their CP and CIP sets in $\mathcal{O}(1)$, which allows efficient processing of operations.

The client component runs at each user's local site inside the web browser. It utilizes *TypeScript*, a type-safe superset of JavaScript [BAT14], to provide a local feature modeling environment. Each client is connected via a *WebSocket* to a central variED server. The WebSocket protocol [FM11] is well-suited for our application because it provides an ordered, stable and permanent communication channel required for causality preservation (cf. Section 5.4). In addition, WebSockets allow to receive operations from the server at any time which facilitates real-time oper-



(a) Desktop (b) Mobile

Figure 6.2: variED running on different devices.

ation propagation. The client further provides a responsive feature modeling user interface (shown in Figure 6.2), which was built with Facebook's user interface library *React* [HOSC16]. Further, we utilize the data visualization framework *D3.js* to draw feature diagrams [BOH11]. Notably, D3.js allows us to animate feature model changes, such as created or removed features. This is a first measure to raise awareness of other collaborators' actions in our prototype.

By compiling the collaboration kernel to JavaScript with *ClojureScript* [McG11], the client can seamlessly use the *generate* and *receive* algorithms required to process operations. Between the collaboration kernel and the user interface, the *Redux* library orchestrates state management and decouples all business logic from the user interface [BP17]. By adopting React and Redux, we follow modern industry practices and aim to make our prototype ready for future extensions [BGR18].

The server component is executed on a central server, which all clients connect to. It is written in Java and runs on standard servlet containers such as *Apache Tomcat* or *Jetty* [HC01]. Instead of a user interface, it includes a client manager which allows clients to join and leave collaboration sessions. By relying on the FeatureIDE core library, we enable users to load any feature model that is compatible with FeatureIDE. Further, because Clojure natively runs on the Java Virtual Machine, the server can simply call the collaboration kernel's *forward* algorithm using standard interop facilities.

To allow for practical usage, we have deployed and tested variED on various cloud infrastructure providers, e.g., *Heroku*, *Amazon AWS*, and *Microsoft Azure* (see our GitHub repository). In addition, we developed integration and unit test suites for the collaboration kernel and client components. We are positive that more comprehensive test suites, together with our correctness argumentation in Section 5.4, can adequately assure the overall correctness of our prototype.

7. Related Work

In this thesis, we proposed a concurrency control technique for collaborative real-time feature modeling. We are not aware of any other work that specifically addresses optimistic concurrency control for feature modeling. However, previous research has investigated collaboration and evolution on feature models and SPLs in general. In this chapter, we present selected publications and how they relate to our work.

Prior work on feature model editing has mostly focused on the single-user case. Popular tools include *FeatureIDE* [MTS+17], *pure::variants* [Beu08], *Gears* [Kru07], *S.P.L.O.T.* [MBC09], and *FAMILIAR* [ACLF13]; others are reviewed by Alves Pereira et al. [APCF14]. To the best of our knowledge, none of these tools support real-time collaboration; rather, they allow asynchronous collaboration using version control systems. While S.P.L.O.T. allows for online usage, its operation model is pessimistic and does not support collaboration. FAMILIAR allows to merge feature models from different collaborators, but does not handle any emerging conflicts.

Closest to our work is the *CoFM* environment proposed by Yi et al. [YZZJ12]. CoFM allows collaboration between stakeholders to construct a shared feature model and allows them to evaluate other users' work by selecting and denying model elements. Every collaborator then has a personal view that reflects her opinions about the domain. Our work differs from CoFM in several ways: First, we only construct one shared feature model, and model elements are not evaluated in hindsight. Second, we describe in detail how to detect and resolve conflicting operations, while CoFM denies any operation that might fail, therefore violating intention preservation. Third, we provide an optimistic concurrency control technique to ensure responsiveness and unconstrained collaboration, whereas CoFM employs a pessimistic client/server architecture [YZZ+10].

In the literature, several approaches for collaborative product configuration have been proposed [HHS+13, MC10, VGPC18, ZLZ+13, ZYZJ13]. These approaches differ from our work in that they do not address feature modeling and their operation model is pessimistic. Further, detection of conflicts in extended feature models has been investigated [DKL+16, LSW15, OGRT15]. In contrast to our work, these

approaches focus on ensuring semantic, not syntactic properties of feature models and configurations, and do not consider collaborative editing.

Linsbauer et al. [LBG17] classify variation control systems, which allow projectional editing on SPL artifacts. In many tools, they notice a lack of collaboration support compared to regular version control systems. Stănciulescu et al. [SBWW16] note that the projectional editing workflow of variation control systems may be unfamiliar and requires mental effort. This is in contrast to our approach, which aims at a simple and intuitive usage without requiring any background knowledge. Schwägerl and Westfechtel [SW17] introduce the *SuperMod* variation control system for filtered editing in model-driven SPLs. They consider consistency problems concerning relationships between the variability model, configurations, and scope of changes. They further extend their approach with support for asynchronous multi-user collaboration [SW16]. However, their approach neither supports real-time editing, nor addresses conflicts that emerge from changes submitted by multiple collaborators.

Feature model evolution has been investigated in the literature, although without focus on collaboration [EBLSP10, McG03]. In particular, several techniques for *change impact analysis* have been proposed [CGC+11, HGBS18, MBBA16, PDŠ12], which are used to measure and evaluate conflict potential for configuration and modeling decisions. Although not explicitly addressing collaboration, these techniques might be incorporated in collaborative feature modeling and product configuration to facilitate understanding and resolution of conflicts. *EvoFM* is an approach for modeling variability over time proposed by Botterweck et al. [BPD+10]. They include a catalogue of evolution operators, which resemble our compound operations discussed in Section 3.2, but they do not address collaboration. Nieke et al. [NSS16] introduce *temporal feature models* to ensure valid configurations as an SPL evolves. Their approach detects inconsistencies between a feature model and its configurations as well as evolution paradoxes [NST18], but they do not propose a conflict resolution strategy other than rejecting problematic changes.

Zaid et al. [ZKDT09] propose to employ *semantic web* technology, i.e., OWL ontologies, to unify feature model representations and integrate segmented feature models. Their approach includes a conflict detection mechanism; in contrast to our work, no editing facilities are provided, and it is not clear how concurrent modifications can be made to a single feature model segment.

Masson et al. [MCS17] propose a feature model for collaborative modeling environments, which is intended to guide the development and analysis of such environments. In the context of their work, our concept may be classified as an optimistic, operation-based, real-time, remote editor with manual conflict resolution.

8. Conclusion

Software product lines are a systematic approach to manage complexity in highly configurable software systems. The customizability of an SPL is typically captured in a feature model, which encodes characteristics of the software, so-called features, as well as dependencies between such features. Constructing and maintaining such a feature model is a critical activity in SPL engineering that affects the entire SPL development process. Further, feature models encode the entire domain knowledge relevant to the SPL, which is commonly spread across different stakeholders. Thus, collaboration among stakeholders is vital in the feature modeling process.

However, limited tooling and the vague nature of natural language currently hamper collaborative editing of feature models. Existing version and variation control systems only address asynchronous collaboration, although we see several use cases for collaborative feature modeling in real time. In particular, collaborative real-time feature modeling systems provide instant feedback from other collaborators, which alleviates the potential for divergence. Thus, engineers can collaborate more tightly and take advantage of their shared domain knowledge.

Contributions

In this thesis, we contributed a novel concept for collaborative real-time feature modeling. We focused on the technical aspect of consistency maintenance to ensure syntactic and semantic consistency of the edited feature model at all times. To this end, we first introduced a feature model representation and operation model suitable for collaboration in Chapter 3. We then identified several requirements for a suitable concurrency control technique in Chapter 4. For our concept, we chose the Multi-Version Multi-Display technique, which allows optimistic operation generation while preserving all user intentions. In Chapter 5, we adjusted the MVMD technique to allow for collaborative feature modeling, focusing on detection and resolution of conflicts. Thus, we answered the research questions posed in Chapter 1. Finally, we justified the correctness of our approach and, in Chapter 6, presented a first implementation of our concept.

We conclude the thesis with a brief discussion of our concept's key characteristics, pointing out what can and cannot be expected of our concept.

- Like groupware in general, our concept is aimed at small groups of collaborators (e.g., 3–4). We assume that collaborators are cooperative and do not intentionally engage in any destructive behavior, so that conflicts emerge only by accident. Thus, we provide an optimistic user interface that allows for fast and responsive editing of feature models.
- In our concept, divergence is prevented because users are instantly prompted to resolve conflicts (cf. Section 5.2). Thus, we aim to reduce confusion introduced by complicated merge conflicts, which are common in version control systems.
- Due to the optimistic nature of our editor, implementations may allow users to continue editing when they are offline or have an unstable internet connection. However, we recommend keeping these offline periods as short as possible (e.g., a few minutes) to allow for frequent synchronization and garbage collection. For longer offline periods, a version control system should be used instead.
- For developers, we offer several variation points to adapt our concept to different environments (corresponding to the flexibility requirement in Section 4.1): First, our concept is designed to allow for additional feature model representations, operations, and semantic properties. Second, it supports both client/server and peer-to-peer topologies. For our prototype, we choose a client/server architecture, but peer-to-peer may be employed for ad-hoc modeling sessions or when a central server is inappropriate due to privacy policies. Third, while our concept is optimistic by default, it may also be implemented in a pessimistic fashion. That is, the MOVIC algorithm might only run on a central server, removing the need for a collaboration kernel on every client. Although this compromises on the responsiveness of the editor, it facilitates implementation and allows to run the editor on end devices with constrained resources, such as CPU, RAM, or battery power. In such a system, users might choose for themselves whatever level of optimism suits their preferences best.
- In Section 5.4, we inferred that our concept is consistent according to the CCI model. However, we cannot guarantee feature model consistency regarding arbitrary semantic properties (such as dead feature analysis) in the case of n-wise conflicts. Because detecting such conflicts is computationally difficult, we leave their solution to the collaborators.

Apart from our concept for collaborative real-time feature modeling, we also make some general contributions to optimistic concurrency control with our work. In particular, we extend the Multi-Version Multi-Display technique with support for arbitrary interrelated objects and semantic consistency properties (cf. Section 5.1), as suggested by Sun and Chen [SC02]. Further, we address future directions from Sun and Ellis [SE98] by proposing an extensible two-layered operation architecture (cf. Section 3.2) and an adaptable process for finding group consensus (cf. Section 5.2). We believe that these techniques may also be useful in editing domains other than feature modeling, as indicated in the next section.

8.1 Future Work

In the following, we list opportunities we identified for future research.

In this thesis, we focused on concurrency control and the detection and resolution of conflicts when they emerge. This is necessary to maintain consistency as per the CCI model, however, it is desirable to avoid conflicts in the first place, if possible. Thus, past research has identified techniques to raise *awareness* among collaborators, e.g., by displaying a collaborator's current location in a text editor [GG02]. This raises the question how such awareness techniques may be adapted to collaborative feature modeling to reduce the probability for conflict.

The concept we presented is tailored to collaborative feature modeling, i.e., modeling the variability of an SPL. In Chapter 7, we presented related work on collaborative *product configuration*, i.e., the construction of a valid product by selecting desired features. Compared to feature models, we expect more conflicts to occur in collaborative product configuration, as users may have opposing goals for the configured product. Thus, it remains to be seen whether our approach is also applicable to product configuration or can be combined sensibly with an existing approach.

Schwägerl and Westfechtel [SW16] note that the concurrency control problem addressed in our work is orthogonal to their work on variation control. We believe this to be true, and we think that our real-time concurrency control approach may complement asynchronous variation control systems, such as *SuperMod*, similar to how pair programming complements version control systems: That is, a collaborative real-time editor may be employed for short editing sessions involving tight collaboration; while a variation control system may manage the long-term evolution of an SPL. Future work may address how to integrate these two approaches, so that collaborators can use the tool whose mode of collaboration suits their particular needs best.

Regarding conflict resolution, we currently do not provide guidance for collaborators regarding which version to choose: Collaborators must review the feature model versions manually, then make their decision solely based on the detected conflicts. Instead, we may provide guidance for their decision: For example, the system could compute whether a feature model version only includes refactoring, generalization or specialization operations according to Thüm et al. [TBK09], so that collaborators can assess how conflicting operations affect the modeled SPL. To this end, the system could also employ dedicated *change impact analyses* (cf. Chapter 7).

In the future, we plan to extend our collaborative feature model editor with several valuable features: First, as we value intuitive use and confidence in our system, a facility to *undo* and *redo* operations is desirable, because users expect to be able to reverse an erroneous change. We plan to extend our concept with support for undo/redo by using the basic idea of concurrent inverse operations, which has been proposed by Sun [Sun03]. Second, our editor currently does not support reordering features and cross-tree constraints due to our feature model representation (cf. Section 3.1). However, this may be desired by collaborators, as it is also supported by single-user feature model editors such as FeatureIDE and pure::variants. We therefore plan to extend our definition of features and cross-tree constraints to include

a *position* attribute that specifies their order and add according conflict detection rules to SYNTACTICALLYCONFLICTING. Finally, we may extend our editor with support for other kinds of feature models (e.g., extended feature models) or additional modeling operations (e.g., extracting or merging features).

Finally, we observed in Chapter 5 that the outer conflict relation $\otimes_O$ does not refer to any concepts specific to feature modeling. Thus, we believe that our concept is generalizable to other modeling domains with clear semantics for documents (e.g., UML and other modeling languages). To this end, we aim to find document and operation models that are suitable for modeling environments in general. The inner conflict relation $\otimes_I$ may then employ a domain-specific set of conflict detection rules. It remains an open question how to assist developers in finding and verifying such conflict detection rules.

Bibliography

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013. (cited on Page 1, 5, 6, 7, 8, and 65)

[ACLF13] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming*, 78(6):657–681, 2013. (cited on Page 83)

[AGU72] Alfred V. Aho, Michael R. Garey, and Jeffrey D. Ullman. The Transitive Reduction of a Directed Graph. *SIAM Journal on Computing*, 1(2):131–137, 1972. (cited on Page 55)

[APCF14] Juliana Alves Pereira, Kattiana Constantino, and Eduardo Figueiredo. A Systematic Literature Review of Software Product Line Management Tools. In *Software Reuse for Dynamic Systems in the Cloud and Beyond*, pages 73–89. Springer, 2014. (cited on Page 83)

[ARW+13] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Grösslinger, and Dirk Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proceedings of the International Conference on Software Engineering*, pages 482–491. IEEE, 2013. (cited on Page 65)

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference*, pages 7–20. Springer, 2005. (cited on Page 7 and 19)

[BAT14] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014. (cited on Page 81)

[BBA98] Sumeer Bhola, Guruduth Banavar, and Mustaque Ahamad. Responsiveness and Consistency Tradeoffs in Interactive Groupware. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 79–88. ACM, 1998. (cited on Page 38)

[BBS10] Jan Bosch and Petra M. Bosch-Sijtsema. Softwares Product Lines, Global Development and Ecosystems: Collaboration in Software Engineering. In Ivan Mistrík, John Grundy, André Hoek, and Jim White-

head, editors, *Collaborative Software Engineering*, chapter 4, pages 77–92. Springer, 2010. (cited on Page 1)

[BDC+89] T. F. Bowen, F. S. Dworack, C. H. Chow, N. Griffeth, G. E. Herman, and Y.-J. Lin. The Feature Interaction Problem in Telecommunications Systems. In *Proceedings of the International Conference on Software Engineering for Telecommunication Switching Systems*, pages 59–62. IET, 1989. (cited on Page 63)

[Ber12] Gary Bernhardt. Functional Core, Imperative Shell. https://www.destroyallsoftware.com/screencasts/catalog/functional-core-imperative-shell, 2012. Accessed March 18, 2019. (cited on Page 80)

[Beu08] Danilo Beuche. Modeling and Building Software Product Lines with Pure::Variants. In *Proceedings of the International Software Product Line Conference*, pages 358–358, 2008. (cited on Page 2, 26, and 83)

[BG93] Thomas Berlage and Andreas Genau. A Framework for Shared Applications with a Replicated Architecture. In *Proceedings of the Symposium on User Interface Software and Technology*, pages 249–257. ACM, 1993. (cited on Page 41)

[BGBG95] Ronald M. Baecker, Jonathan Grudin, William A. S. Buxton, and Saul Greenberg. *Human-Computer Interaction: Toward the Year 2000*. Morgan Kaufmann, 1995. (cited on Page 9 and 10)

[BGR+17] Rabih Bashroush, Muhammad Garba, Rick Rabiser, Iris Groher, and Goetz Botterweck. CASE Tool Support for Variability Management in Software Product Lines. *ACM Computing Surveys*, 50(1):14:1–14:45, 2017. (cited on Page 8)

[BGR18] Raphaël Benitte, Sacha Greif, and Michael Rambeau. The State of JavaScript. https://2018.stateofjs.com, 2018. Accessed March 18, 2019. (cited on Page 82)

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. (cited on Page 16)

[BMG17] Andreas Biørn-Hansen, Tim A. Majchrzak, and Tor-Morten Grønli. Progressive Web Apps: The Possible Web-Native Unifier for Mobile Development. In *International Conference on Web Information Systems and Technologies*, pages 344–351. SciTePress, 2017. (cited on Page 79)

[BNR+14] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. Three Cases of Feature-Based Variability Modeling in Industry. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, pages 302–319. Springer, 2014. (cited on Page 1)

[BOH11] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. $\mathbb{D}^3$: Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011. (cited on Page 82)

[BP17] Alex Banks and Eve Porcello. *Learning React: Functional Web Development with React and Redux*. O'Reilly Media, 2017. (cited on Page 82)

[BPD+10] Goetz Botterweck, Andreas Pleuss, Deepak Dhungana, Andreas Polzer, and Stefan Kowalewski. EvoFM: Feature-Driven Planning of Product-Line Evolution. In *Proceedings of the International Workshop on Product Line Approaches in Software Engineering*, pages 24–31. ACM, 2010. (cited on Page 84)

[BRN+13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A Survey of Variability Modeling in Industrial Practice. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems*, pages 7:1–7:8. ACM, 2013. (cited on Page 6, 7, 8, and 19)

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–636, 2010. (cited on Page 8 and 65)

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated Reasoning on Feature Models. In *Proceedings of the International Conference on Advanced Information Systems Engineering*, pages 491–503. Springer, 2005. (cited on Page 8 and 19)

[BvdBB01] Rafael Bidarra, Eelco van den Berg, and Willem F. Bronsvoort. Collaborative Modeling with Features. In *Proceedings of the Design Engineering Technical Conference and Computers and Information in Engineering Conference*. ASME, 2001. (cited on Page 38)

[Bö04] Günter Böckle. *Software-Produktlinien: Methoden, Einführung und Praxis*. dpunkt-Verlag, 2004. (cited on Page 6)

[Cam00] Jeffrey Dennis Campbell. *Consistency Maintenance for Real-Time Collaborative Diagram Development*. PhD thesis, University of Pittsburgh, 2000. (cited on Page 38)

[CB11] Lianping Chen and Muhammad Ali Babar. A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. *Information and Software Technology*, 53(4):344–362, 2011. (cited on Page 6)

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. (cited on Page 6 and 7)

[CGC+11] Hyun Cho, Jeff Gray, Yuanfang Cai, Sonny Wong, and Tao Xie. Model-Driven Impact Analysis of Software Product Lines. In Janis Osis and Erika Asnina, editor, *Model-Driven Domain Analysis and Software Development: Architectures and Functions*, chapter 13, pages 275–303. IGI Global, 2011. (cited on Page 84)

[CGR+12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems*, pages 173–182. ACM, 2012. (cited on Page 6)

[Cha09] Scott Chacon. *Pro Git*. Springer, 2009. (cited on Page 17)

[Che01] David Chen. *Consistency Maintenance in Collaborative Graphics Editing Systems*. PhD thesis, Griffith University, 2001. (cited on Page 16, 17, 20, 25, 27, 44, 46, 48, and 73)

[CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Using Feature Models. In *Proceedings of the International Software Product Line Conference*, pages 266–283. Springer, 2004. (cited on Page 8 and 19)

[CK02] Paul Clements and Charles Krueger. Point/Counterpoint. *IEEE Software*, 19(4):28–31, 2002. (cited on Page 5)

[CKMRM03] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41(1):115–141, 2003. (cited on Page 63)

[CN02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002. (cited on Page 1 and 5)

[Coc05] Alistair Cockburn. Hexagonal Architecture. http://web.archive.org/web/20180822100852/https://alistair.cockburn.us/Hexagonal+architecture, 2005. Accessed March 18, 2019. (cited on Page 80)

[Cor95a] Gordon V. Cormack. A Calculus for Concurrent Update. Technical Report CS-95-06, University of Waterloo, 1995. (cited on Page 27)

[Cor95b] Gordon V. Cormack. A Counterexample to the Distributed Operational Transform and a Corrected Algorithm for Point-to-Point Communication. Technical Report CS-95-08, University of Waterloo, 1995. (cited on Page 44)

[CS99] Peter H. Carstensen and Kjeld Schmidt. Computer Supported Cooperative Work: New Challenges to Systems Design. In K. Itoh, editor, *Handbook of Human Factors*, pages 619–636. 1999. (cited on Page 9)

[CS01a] David Chen and Chengzheng Sun. Optional Instant Locking in Distributed Collaborative Graphics Editing Systems. In *Proceedings of*

the International Conference on Parallel and Distributed Systems, pages 109–116. IEEE, 2001. (cited on Page 40)

[CS01b] David Chen and Chengzheng Sun. Undoing Any Operation in Collaborative Graphics Editing Systems. In *Proceedings of the International Conference on Supporting Group Work*, pages 197–206. ACM, 2001. (cited on Page 23)

[DCS94] Prasun Dewan, Rajiv Choudhary, and Honghai Shen. An Editing-Based Characterization of the Design Space of Collaborative Applications. *Journal of Organizational Computing*, 4(3):219–239, 1994. (cited on Page 10, 17, 37, 38, and 40)

[DDIZ18] Gabriele D'Angelo, Angelo Di Iorio, and Stefano Zacchiroli. Spacetime Characterization of Real-Time Collaborative Editing. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):41:1–41:19, 2018. (cited on Page 10 and 38)

[Dew99] Prasun Dewan. Architectures for Collaborative Applications. *Computer Supported Co-Operative Work*, 7:169–193, 1999. (cited on Page 12)

[DKL+16] Frederik Deckwerth, Géza Kulcsár, Malte Lochau, Gergely Varró, and Andy Schürr. Conflict Detection for Edits on Extended Feature Models using Symbolic Graph Transformation. *Electronic Proceedings in Theoretical Computer Science*, 206:17–31, 2016. (cited on Page 83)

[DPN98] Alan R. Dennis, Sridar K. Pootheri, and Vijaya L. Natarajan. Lessons from the Early Adopters of Web Groupware. *Journal of Management Information Systems*, 14(4):65–86, 1998. (cited on Page 69)

[EBLSP10] Christoph Elsner, Goetz Botterweck, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Variability in Time — Product Line Variability and Evolution Revisited. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems*, pages 131–137, 2010. (cited on Page 84)

[EG89] Clarence Ellis and Simon Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the International Conference on Management of Data*, pages 399–407. ACM, 1989. (cited on Page 10, 11, 13, 17, 25, 42, 43, and 44)

[EGR91] Clarence Ellis, Simon Gibbs, and Gail Rein. Groupware: Some Issues and Experiences. *Communications of the ACM*, 34(1):39–58, 1991. (cited on Page 9, 37, 38, and 39)

[EN10] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 2010. (cited on Page 40)

[FBGR13] Alexander Felfernig, David Benavides, José A. Galindo, and Florian Reinfrank. Towards Anomaly Explanation in Feature Models. In

Proceedings of the International Configuration Workshop, pages 117–124, 2013. (cited on Page 8 and 65)

[Fid88] Colin J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. *Australian Computer Science Communications*, 10(1):56–66, 1988. (cited on Page 74 and 81)

[FLLE14] Shannon Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *Proceedings of the International Conference on Software Maintenance and Evolution*, pages 391–400. IEEE, 2014. (cited on Page 1)

[FM11] Ian Fette and Alexey Melnikov. The WebSocket Protocol. RFC 6455, IETF, 2011. (cited on Page 81)

[Fra09] Neil Fraser. Differential Synchronization. In *Proceedings of the Symposium on Document Engineering*, pages 13–20. ACM, 2009. (cited on Page 17)

[GG02] Carl Gutwin and Saul Greenberg. A Descriptive Framework of Workspace Awareness for Real-Time Groupware. *Computer Supported Cooperative Work*, 11(3):411–446, 2002. (cited on Page 87)

[Gib89] Simon Gibbs. LIZA: An Extensible Groupware Toolkit. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 29–35. ACM, 1989. (cited on Page 69)

[GK09] Tom Gross and Michael Koch. *Computer-Supported Cooperative Work*. Oldenbourg Verlag, 2009. (cited on Page 9)

[GLM11] Max Goldman, Greg Little, and Robert C. Miller. Real-Time Collaborative Coding in a Web IDE. In *Proceedings of the Symposium on User Interface Software and Technology*, pages 155–164. ACM, 2011. (cited on Page 17 and 79)

[GM94] Saul Greenberg and David Marwood. Real Time Groupware as a Distributed System: Concurrency Control and itss Effect on the Interface. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 207–217. ACM, 1994. (cited on Page 12, 25, 37, 38, 40, and 41)

[GR99] Saul Greenberg and Mark Roseman. Groupware Toolkits for Synchronous Work. In Michel Beaudouin-Lafon, editor, *Computer Supported Co-Operative Work*, chapter 6, pages 135–168. Wiley, 1999. (cited on Page 12)

[Gre91] Saul Greenberg. Personalizable Groupware: Accommodating Individual Roles and Group Differences. In *Proceedings of the European Conference on Computer-Supported Cooperative Work*, pages 17–31. Springer, 1991. (cited on Page 39)

[Gru94] Jonathan Grudin. Computer-Supported Cooperative Work: History and Focus. *Communications of the ACM*, 27(5):19–26, 1994. (cited on Page 9)

[HC01] Jason Hunter and William Crawford. *Java Servlet Programming: Help for Server Side Java Developers*. O'Reilly Media, 2001. (cited on Page 82)

[HGBS18] Ines Hajri, Arda Goknil, Lionel C. Briand, and Thierry Stephany. Change Impact Analysis for Evolving Configuration Decisions in Product Line Use Case Models. *Journal of Systems and Software*, 139:211–237, 2018. (cited on Page 84)

[HHS+13] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, Dirk Deridder, and Ebrahim Khalil Abbasi. Supporting Multiple Perspectives in Feature-Based Configuration. *Software & Systems Modeling*, 12(3):641–663, 2013. (cited on Page 83)

[Hic08] Rich Hickey. The Clojure Programming Language. In *Proceedings of the Symposium on Dynamic Languages*. ACM, 2008. (cited on Page 80)

[HOSC16] Pete Hunt, Paul O'Shannessy, Dave Smith, and Terry Coatta. React: Facebook's Functional Turn on Writing JavaScript. *ACM Queue*, 14(4):40:96–40:112, 2016. (cited on Page 82)

[IROM06] Abdessamad Imine, Michaël Rusinowitch, Gérald Oster, and Pascal Molli. Formal Design and Verification of Operational Transformation Algorithms for Copies Convergence. *Theoretical Computer Science*, 351(2):167–183, 2006. (cited on Page 27 and 39)

[JGH07] Caroline Jay, Mashhuda Glencross, and Roger Hubbold. Modeling the Effects of Delayed Haptic and Visual Feedback in a Collaborative Virtual Environment. *ACM Transactions on Computer-Human Interaction*, 14(2), 2007. (cited on Page 12)

[JLK17] Young-Wook Jung, Youn-Kyung Lim, and Myung-Suk Kim. Possibilities and Limitations of Online Document Tools for Design Collaboration: The Case of Google Docs. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 1096–1108. ACM, 2017. (cited on Page 2)

[Joh91] Robert Johansen. Groupware: Future Directions and Wild Cards. *Journal of Organizational Computing*, 1(2):219–227, 1991. (cited on Page 9)

[Kah62] A. B. Kahn. Topological Sorting of Large Networks. *Communications of the ACM*, 5(11):558–562, 1962. (cited on Page 60)

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA)

Feasibility Study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon University, 1990. (cited on Page 1, 6, and 7)

[KFM+16] Jacob Krüger, Wolfram Fenske, Jens Meinicke, Thomas Leich, and Gunter Saake. Extracting Software Product Lines: A Cost Estimation Perspective. In *Proceedings of the International Software Product Line Conference*, pages 354–361. ACM, 2016. (cited on Page 5)

[KMB+01] Peter Knauber, Jesús Bermejo Muñoz, Günter Böckle, Julio Cesar Sampaio do Prado Leite, Frank van der Linden, Linda Northrop, Michael Stark, and David M. Weiss. Quantifying Product Line Benefits. In *Proceedings of the International Workshop on Software Product-Family Engineering*, pages 155–163. Springer, 2001. (cited on Page 1 and 5)

[Kru07] Charles W. Krueger. BigLever Software Gears and the 3-Tiered SPL Methodology. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 844–845. ACM, 2007. (cited on Page 2 and 83)

[KTM+17] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. Is There a Mismatch Between Real-World Feature Models and Product-Line Research? In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 291–302. ACM, 2017. (cited on Page 8)

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978. (cited on Page 15 and 42)

[LBG17] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. A Classification of Variation Control Systems. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 49–62. ACM, 2017. (cited on Page 2 and 84)

[LCD+07] Kai Lin, David Chen, Geoff Dromey, Stephen Xia, and Chengzheng Sun. API Design Recommendations for Facilitating Conversion of Single-User Applications into Collaborative Applications. In *Proceedings of the International Conference on Collaborative Computing*, pages 309–317. IEEE, 2007. (cited on Page 20 and 25)

[LFST07] Andrea De Lucia, Fausto Fasano, Giuseppe Scanniello, and Genoveffa Tortora. Enhancing Collaborative Synchronous UML Modelling with Fine-Grained Versioning of Software Artefacts. *Journal of Visual Languages & Computing*, 18(5):492–503, 2007. (cited on Page 38)

[LHB01] Roberto E. Lopez-Herrejon and Don Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *Generative and Component-Based Software Engineering*, pages 10–24. Springer, 2001. (cited on Page 7)

[LK04] John M. Linebarger and G. Drew Kessler. Concurrency Control Mechanisms for Closely Coupled Collaboration in Multithreaded Peer-to-Peer Virtual Environments. *Presence: Teleoperators and Virtual Environments*, 13(3):296–314, 2004. (cited on Page 41)

[LLFW05] W. D. Li, W. F. Lu, J. Y. H. Fuh, and Y. S. Wong. Collaborative Computer-Aided Design — Research and Development Status. *Computer-Aided Design*, 37(9):931–940, 2005. (cited on Page 10)

[LMS05] Paul J. Leach, Michael Mealling, and Rich Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, Network Working Group, 2005. (cited on Page 81)

[LSW15] Uwe Lesta, Ina Schaefer, and Tim Winkelmann. Detecting and Explaining Conflicts in Attributed Feature Models. *Electronic Proceedings in Theoretical Computer Science*, 182:31–43, 2015. (cited on Page 83)

[Mat88] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. North Holland, 1988. (cited on Page 15, 55, and 81)

[MBBA16] Jihen Maâzoun, Nadia Bouassida, and Hanêne Ben-Abdallah. Change Impact Analysis for Software Product Lines. *Journal of King Saud University — Computer and Information Sciences*, 28(4):364–380, 2016. (cited on Page 84)

[MBC09] Marcilio Mendonça, Moises Branco, and Donald Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 761–762. ACM, 2009. (cited on Page 83)

[MC10] Marcilio Mendonça and Donald Cowan. Decision-Making Coordination and Efficient Reasoning Techniques for Feature-Based Configuration. *Science of Computer Programming*, 75(5):311–332, 2010. (cited on Page 83)

[McG03] John D. McGregor. The Evolution of Product Line Assets. Technical Report CMU/SEI-2003-TR-005, Carnegie Mellon University, 2003. (cited on Page 84)

[McG11] Mark McGranaghan. ClojureScript: Functional Programming for JavaScript Platforms. *IEEE Internet Computing*, 15(6):97–102, 2011. (cited on Page 82)

[MCS17] Constantin Masson, Jonathan Corley, and Eugene Syriani. Feature Model for Collaborative Modeling Environments. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, pages 164–173, 2017. (cited on Page 84)

[MD96] Jonathan Munson and Prasun Dewan. A Concurrency Control Framework for Collaborative Systems. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 278–287. ACM, 1996. (cited on Page 40)

[MMvM+95] Thomas P. Moran, Kim McCall, Bill van Melle, E. Ronby Pedersen, and Frank Halasz. Some Design Principles for Sharing in Tivoli, a Whiteboard Meeting-Support Tool. In Saul Greenberg, Stephen Hayne, and R. Rada, editors, *Groupware for Real-Time Drawings: A Designer's Guide*, pages 24–36. McGraw-Hill, 1995. (cited on Page 46)

[MO92] Lola J. McGuffin and Gary M. Olson. *ShrEdit: A Shared Electronic Workspace*. University of Michigan, Cognitive Science and Machine Intelligence Laboratory, 1992. (cited on Page 40)

[MRS+04] Meredith Ringel Morris, Kathy Ryall, Chia Shen, Clifton Forlines, and Frederic Vernier. Beyond "Social Protocols": Multi-user Coordination Policies for Co-Located Groupware. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 262–265. ACM, 2004. (cited on Page 69)

[MTS+17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017. (cited on Page 2, 19, 26, and 83)

[MWC09] Marcilio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. SAT-Based Analysis of Feature Models is Easy. In *Proceedings of the International Software Product Line Conference*, pages 231–240. Carnegie Mellon University, 2009. (cited on Page 41)

[NCDL95] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System. In *Proceedings of the Symposium on User Interface Software and Technology*, pages 111–120. ACM, 1995. (cited on Page 11, 43, and 44)

[NES17] Michael Nieke, Gil Engel, and Christoph Seidl. DarwinSPL: An Integrated Tool Suite for Modeling Evolving Context-Aware Software Product Lines. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems*, pages 92–99. ACM, 2017. (cited on Page 8)

[NL91] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *The Computer Journal*, 24(8):52–60, 1991. (cited on Page 17)

[NSS16] Michael Nieke, Christoph Seidl, and Sven Schuster. Guaranteeing Configuration Validity in Evolving Software Product Lines. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems*, pages 73–80. ACM, 2016. (cited on Page 8 and 84)

[NST18] Michael Nieke, Christoph Seidl, and Thomas Thüm. Back to the Future: Avoiding Paradoxes in Feature-Model Evolution. In *Proceedings of the International Software Product Line Conference*, pages 48–51. ACM, 2018. (cited on Page 84)

[OGRT15] Lina Ochoa, Oscar González-Rojas, and Thomas Thüm. Using Decision Rules for Solving Conflicts in Extended Feature Models. In *Proceedings of the International Conference on Software Language Engineering*, pages 149–160. ACM, 2015. (cited on Page 83)

[OMUI06] Gérald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems. In *Proceedings of the International Conference on Collaborative Computing*, pages 1–10. IEEE, 2006. (cited on Page 23)

[O'S09] Bryan O'Sullivan. Making Sense of Revision-Control Systems. *ACM Queue*, 7(7):30:30–30:40, 2009. (cited on Page 2 and 17)

[Pap02] Constantinos Papadopoulos. A Multiple Granularity Locking Protocol for CSCW. *International Journal of Cooperative Information Systems*, 11(1&2):21–50, 2002. (cited on Page 40)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, 2005. (cited on Page 1, 5, and 6)

[PCSF08] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion: Next Generation Open Source Version Control.* O'Reilly Media, 2008. (cited on Page 17)

[PDŠ12] Paulius Paskevicius, Robertas Damasevicius, and Vytautas Štuikys. Change Impact Analysis of Feature Models. In *Proceedings of the International Conference on Information and Software Technologies*, pages 108–122. Springer, 2012. (cited on Page 84)

[PLP11] Richard Pohl, Kim Lauenroth, and Klaus Pohl. A Performance Comparison of Contemporary Algorithmic Approaches for Automated Analysis Operations on Feature Models. In *Proceedings of the International Conference on Automated Software Engineering*, pages 313–322. IEEE, 2011. (cited on Page 65)

[Pra99] Atul Prakash. Group Editors. In Michel Beaudouin-Lafon, editor, *Computer Supported Co-Operative Work*, chapter 5, pages 103–134. Wiley, 1999. (cited on Page 2, 10, 12, and 38)

[RBIQ13] Aurel Randolph, Hanifa Boucheneb, Abdessamad Imine, and Alejandro Quintero. On Consistency of Operational Transformation Approach. *Electronic Proceedings in Theoretical Computer Science*, 107:45–59, 2013. (cited on Page 39)

[RNRG96] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 288–297. ACM, 1996. (cited on Page 25 and 43)

[SBWW16] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *Proceedings of the International Conference on Software Maintenance and Evolution*, pages 323–333. IEEE, 2016. (cited on Page 84)

[SC00] Chengzheng Sun and David Chen. A Multi-Version Approach to Conflict Resolution in Distributed Groupware Systems. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 316–325. IEEE, 2000. (cited on Page 48)

[SC02] Chengzheng Sun and David Chen. Consistency Maintenance in Real-Time Collaborative Graphics Editing Systems. *ACM Transactions on Computer-Human Interaction*, 9(1):1–41, 2002. (cited on Page 10, 13, 27, 38, 46, 48, 49, 50, 51, 52, 54, 69, 74, 77, 78, and 86)

[Sch01] Rüdiger Schollmeier. 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*, pages 101–102. IEEE, 2001. (cited on Page 10)

[SE98] Chengzheng Sun and Clarence Ellis. Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 59–68. ACM, 1998. (cited on Page 10, 11, 15, 42, 78, and 86)

[SFB+87] Mark Stefik, Gregg Foster, Daniel G. Bobrow, Kenneth Kahn, Stan Lanning, and Lucy Suchman. Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings. *Communications of the ACM*, 30(1):32–47, 1987. (cited on Page 38, 44, and 69)

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *Proceedings of the International Requirements Engineering Conference*, pages 139–148. IEEE, 2006. (cited on Page 7 and 20)

[SJZ+98] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, 1998. (cited on Page 10, 12, 13, 14, 15, 16, 27, 38, 42, 55, 73, 74, 77, and 78)

[SM94] Reinhard Schwarz and Friedemann Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7(3):149–174, 1994. (cited on Page 15 and 55)

[SPBZ11a] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A Comprehensive Study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria — Centre Paris-Rocquencourt, 2011. (cited on Page 41 and 70)

[SPBZ11b] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011. (cited on Page 11, 23, and 41)

[SRC+12] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software Diversity: State of the Art and Perspectives. *International Journal on Software Tools for Technology Transfer*, 14(5):477–495, 2012. (cited on Page 6)

[SS99] Chengzheng Sun and Roc Sosič. 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*, pages 15–22. IEEE, 1999. (cited on Page 44 and 74)

[Sta19] StatCounter Global Stats. Desktop vs. Mobile vs. Tablet Market Share Worldwide. http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet, accessed March 17, 2019. (cited on Page 79)

[Sun02] Chengzheng Sun. Optional and Responsive Fine-Grain Locking in Internet-Based Collaborative Systems. *IEEE Transactions on Parallel and Distributed Systems*, 13(9):994–1008, 2002. (cited on Page 14 and 40)

[Sun03] Chengzheng Sun. Undo As Concurrent Inverse in Group Editors. *ACM Transactions on Computer-Human Interaction*, 10(2):7–8, 2003. (cited on Page 45 and 87)

[SW16] Felix Schwägerl and Bernhard Westfechtel. Collaborative and Distributed Management of Versioned Model-Driven Software Product Lines. In *International Joint Conference on Software Technologies*, pages 83–94. SciTePress, 2016. (cited on Page 84 and 87)

[SW17] Felix Schwägerl and Bernhard Westfechtel. 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*, pages 15–28. SciTePress, 2017. (cited on Page 17 and 84)

[SXA14] Chengzheng Sun, Yi Xu, and Agustina Agustina. Exhaustive Search of Puzzles in Operational Transformation. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 519–529. ACM, 2014. (cited on Page 27, 39, and 44)

[SXS+06] Chengzheng Sun, Steven Xia, David Sun, David Chen, Haifeng Shen, and Wentong Cai. Transparent Adaptation of Single-User Applications for Multi-User Real-Time Collaboration. *ACM Transactions on Computer-Human Interaction*, 13(4):531–582, 2006. (cited on Page 10 and 27)

[SXSC04] David Sun, Steven Xia, Chengzheng Sun, and David Chen. Operational Transformation for Collaborative Word Processing. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 437–446. ACM, 2004. (cited on Page 10, 27, 42, 43, and 45)

[SYZC96] Chengzheng Sun, Yun Yang, Yanchun Zhang, and David Chen. A Consistency Model and Supporting Schemes for Real-Time Cooperative Editing Systems. In *Proceedings of the Australian Computer Science Conference*, pages 582–591, 1996. (cited on Page 15)

[TBK09] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning About Edits to Feature Models. In *Proceedings of the International Conference on Software Engineering*, pages 254–264. IEEE, 2009. (cited on Page 87)

[TCD07] Min Tang, Shang-Ching Chou, and Jin-Xiang Dong. Conflicts Classification and Solving for Collaborative Feature Modeling. *Advanced Engineering Informatics*, 21(2):211–219, 2007. (cited on Page 20, 25, and 38)

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering*, pages 107–119. ACM, 1999. (cited on Page 8)

[VGPC18] Sebastian Velásquez-Guevara, Gilberto Pedraza, and Jaime Chavarriaga. Multi-SPLOT: Supporting Multi-User Configurations with Constraint Programming. In *Proceedings of the International Conference on Applied Informatics*, pages 364–378. Springer, 2018. (cited on Page 83)

[Wil91] Paul Wilson. *Computer Supported Cooperative Work: An Introduction*. Springer, 1991. (cited on Page 9)

[Wul95] Volker Wulf. Negotiability: A Metafunction to Tailor Access to Data in Groupware. *Behaviour & Information Technology*, 14(3):143–151, 1995. (cited on Page 44 and 69)

[WV02] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002. (cited on Page 16)

[WW11] Torben Weis and Arno Wacker. Federating Websites with the Google Wave Protocol. *IEEE Internet Computing*, 15(3):51–58, 2011. (cited on Page 10)

[XO05] Liyin Xue and Mehmet A. Orgun. Locking Without Requesting A Lock: A Consistency Maintenance Mechanism in Internet-Based Real-Time Group Editors. *Journal of Parallel and Distributed Computing*, 65(7):801–814, 2005. (cited on Page 40)

[XOZ03] Liyin Xue, Mehmet Orgun, and Kang Zhang. A Multi-Versioning Algorithm for Intention Preservation in Distributed Real-Time Group Editors. In *Proceedings of the Australasian Computer Science Conference*, pages 19–28. ACS, 2003. (cited on Page 49, 69, and 75)

[YZZ+10] Li Yi, Wei Zhang, Haiyan Zhao, Zhi Jin, and Hong Mei. CoFM: A Web-Based Collaborative Feature Modeling System for Internetware Requirements' Gathering and Continual Evolution. In *Proceedings of the Asia-Pacific Symposium on Internetware*, pages 23:1–23:4. ACM, 2010. (cited on Page 83)

[YZZJ12] Li Yi, Haiyan Zhao, Wei Zhang, and Zhi Jin. CoFM: An Environment for Collaborative Feature Modeling. In *Proceedings of the International Requirements Engineering Conference*, pages 317–318. IEEE, 2012. (cited on Page 2 and 83)

[ZKDT09] Lamia Abo Zaid, Frederic Kleinermann, and Olga De Troyer. Applying Semantic Web Technology to Feature Modeling. In *Proceedings of the Symposium on Applied Computing*, pages 1252–1256. ACM, 2009. (cited on Page 84)

[ZLZ+13] Hongxia Zhang, Rongheng Lin, Hua Zou, Fangchun Yang, and Yao Zhao. The Collaborative Configuration of Service-Oriented Product Lines Based on Evolutionary Approach. In *Proceedings of the International Conference on Services Computing*, pages 751–752. IEEE, 2013. (cited on Page 83)

[ZYZJ13] Wei Zhang, Li Yi, HaiYan Zhao, and Zhi Jin. Feature-Oriented Stigmergy-Based Collaborative Requirements Modeling: An Exploratory Approach for Requirements Elicitation and Evolution Based on Web-Enabled Collective Intelligence. *Science China Information Sciences*, 56(8):1–18, 2013. (cited on Page 2 and 83)

I herewith assure that I wrote the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

Magdeburg, 1st April 2019